Extending Cachegrind:
L2 cache inclusion and TLB measuring

Stavros Kaparelos

Bachelor of Science in Computer Science with Honours
The University of Bath
May 2014

Signed:

Extending Cachegrind: L2 cache inlcusion and TLB measuring

Submitted by Stavros Kaparelos

COPYRIGHT

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Signed:

Abstract

Programmers need to develop hardware conscious software in order to optimise efficiency and increase execution speed. To do that, they need tools that can provide such kind of information. Cachegrind is such a tool. It is a cache profiling tool that provides statistics for level 1 and the last level caches. This project extended Cachegrind, firstly, to include information for L2 cache when 3 levels of cache are present and, secondly, to measure the Translation Lookaside Buffer (TLB). L2 cache inclusion presents the same information as presented in the other cache levels, while measuring the TLB provides a) the number of hits and misses, b) the pages used and the times used and c) per file, per function and, per source code line statistics. Any extension is developed to work in an Intel x86 architecture. Finally, extensions have been tested and results have been checked to be logical, as well as to be along the lines of the expected ones.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my supervisor, prof. James Davenport for the support and guidance he offered throughout this project, without whom the project would not have been possible. I would also like to thank Dr. Jessica Jones for pointing me towards some very useful resources.

Chapter 1

Introduction

1.1 Overview

The speed of delivery of data from memory to processors was always slower than the speed of processors making memory accesses of a program a limit to its execution speed [2]. "Hardware designers have come up with sophisticated memory handling and acceleration techniques — such as CPU caches —, but these techniques cannot work optimally, without some help from the programmer" [12].

Several techniques and tools exist that provide help to the programmer to create hardware conscious and efficient software. Among others, dynamic binary analysis tools serve as system profilers aiming to help developers optimise efficiency and increase speed of their programs. *Valgrind* is a widely-used dynamic binary analysis framework which allows for the creation of tools performing dynamic binary instrumentation [22]. It is used by a variety of large projects such as Firefox, OpenOffice, KDE, Gnome, etc. [22]. *Cachegrind* is a tool that belongs to the Valgrind suite and specialises in cache profiling.

Cachegrind provides information for L1 (Level 1) and LL (Last Level: can either be L2 or L3) caches, and, specifically provides CPU cache hits and misses in general for the whole program, as well as per line of code [10]. Cachegrind is able to provide that kind of information by simulating the CPU caches of a computer either by auto-detecting their characteristics, or by letting the user specify them manually. However, it does not provide any information for L2 cache when 3 levels of cache are detected. Moreover, Cachegrind does not simulate nor consider the *Translation Lookaside Buffer* (TLB), which is a cache of popular address translations and has crucial effect on execution speed. TLB's importance is often neglected and much less attention is paid to it than it should [15].

1.2 Project Aims

Within the above mentioned context, the aim of this project is, firstly, to extend Cachegrind to include L2 cache information when 3 levels of cache are present and, secondly to measure TLB.

L2 cache inclusion should present the same information as presented in L1 and LL caches, while measuring TLB should provide per TLB information for: a) the number of hits and misses, b) the pages used and the times used and c) identified parts in the code that caused many misses.

The current project aims to make any extensions work for Intel x86 architecture and it does not account for other architectures. The reason for that is that each architecture has its own characteristics, thus, implementations cannot overlap. Since Intel x86 is a very commonly used architecture, it was thought that more people would benefit from the extensions, thus, it was chosen.

1.3 Project Motivation

There are three main reasons that turn the problem in hand into an interesting worthy of further analysis problem. First of all, when it will be completed, Cachegrind will provide further information for the CPU caches, since it will include L2 cache. L2 cache information might be helpful for programmers to better understand hits and misses that are caused by their program.

However, the major contribution of this project is considered to be TLB measuring. TLB is a cache of popular virtual-to-physical addresses and its importance is often neglected [15]. By including TLB, this project gives developers a new topic to reason about and develop even faster and more efficient programs. Moreover, it contributes to further research work that is in need of an extended version of Cachegrind to include TLB information, as in [15], where the authors would have liked to measure the TLB, but they were in lack of a tool that could do it as desired.

1.4 Document Outline

The chapters included in the present dissertation, the outline of their contents, as well as the purpose they serve are described below:

1. **Chapter 1 - Introduction**
   This chapter introduces the reader to the problem in hand and the context within the problem takes place. Moreover, it identifies the project motivation, i.e. the reasons that the problem is of particular interest and is worthy of further analysis and study. Finally, it outlines the structure of the present dissertation.

2. **Chapter 2 - Literature Review**
   Literature Review contains all the background information needed to understand this dissertation. It introduces the reader to the concept of caching and explains the main three topics of this dissertation — CPU caches, TLB and Cachegrind — as well as their corresponding technical details that are needed to follow the next chapters.

3. **Chapter 3 - L2 Cache Inclusion**
   L2 cache inclusion explains the notion behind Cachegrind's extension to include L2 cache. It starts by explaining the extension from an intangible point of view (i.e. the idea on how to approach the extension) which, as the chapter progresses, turns to a more tangible one, by transforming those ideas to actual code and thus presenting the whole extension from beginning to end. The chapter, then presents the results of the extension, draws upon the validity of the results and, measures the slowdown that the extension introduced.

4. **Chapter 4 - TLB Measuring**
   TLB measuring contains all information related to the TLB measuring extension. The chapter starts by explaining how does the TLB operate, it then describes the capabilities and characteristics of the TLB simulator, it provides an in depth analysis of the simulator's design and implementation, it presents execution results, it draws upon the validity of the results and, finally, it measures the slowdown introduced by the TLB measuring extension.

5. **Chapter 5 - Conclusion**
   Conclusion critically appraises aspects of the project, presents the difficulties encountered, criticises the techniques used to draw upon the validity of the results, elaborates on the lessons learnt and, finally, identifies future work.

6. **Appendix A - TLB Measuring Source Code**
   Appendix A contains the source code of TLB measuring. The reason that not all Cachegrind's source code is included is because it would occupy a large number of

pages. Therefore, the most interesting part of the code was chosen to be presented here.

Chapter 2

Literature Review

2.1 Introduction

Nowadays, computer programs and hardware tend to be more complex than they used to be. Hardware companies increase their hardware speed and quality and programmers try to take advantage of those changes by developing efficient and hardware-conscious software. Two important factors that line with the above contemplation are CPU cache memory and the Traslation-Lookaside Buffer (TLB). Chapter 2 sheds light upon the underlined notions of caching in general, CPU caches and the TLB. Moreover, it introduces and describes Cachegrind, which is a tool used for cache profiling, and Valgrind, which is the framework Cachegrind has been built on. Furthermore, the technological resources used to extend Cachegrind are described and, finally, the way that testing will take place is explained.

2.2 Caching

Section 2.2 focuses on cache memory. After explaining the general notion of caching, it distinguishes between two basic types of cache memory: a) CPU caches, and b) TLB. In addition, the section presents the general framework within which CPU caches and TLB are understood, as well as how they work in detail.

2.2.1 Cache Memory

Cache memory is a physical part of the hardware which is used to increase speed in program execution. Caching refers to making temporary copies of data that are more likely to be used soon by a component (such as CPU) in a storage area [12]. To do this, cache memory "takes advantage of the phenomenon of 'locality'- the nonrandom behavior of typical programs" [1]. This means that programs spend most of their time executing a small fraction of the code and accessing a small fraction of data. A program has *temporal locality* if elements accessed are likely to be accessed again in a short period of time, and *spatial locality*, if elements close to a location accessed are likely to be accessed within a short period of time [1].

Thus, the notion behind the scenes of a program execution is that the CPU first looks for data in cache and, if they are not found, then tries to access them using the main storage area. If data reside in cache, we say there is a *cache hit*, otherwise there is a *cache miss*. The reason why we use a cache instead of directly looking for data and instructions to the main storage area is that accessing a cache is many times faster.

The idea of cache memory can be illustrated using a simplified example from everyday life. Let us suppose that one makes his food shopping once a week. He buys goods he thinks he will most probably use and then stores them in the fridge and the kitchen selves. When time for cooking a recipe comes one looks in the storage area (i.e. fridge and selves) in order to find the ingredients needed for the recipe. In case they are there, everything is fine and

one can proceed to cooking. Otherwise, he has to go to the supermarket or the food supplier to buy what is missing. The process of having stored several goods in one's house saves him time and speeds up the cooking process, in terms that it will be finished earlier, whereas the time spent to visit the supermarket and come back can be used to do something else, such as listening to music.

The notion of cache memory is exactly the same as saving goods in the fridge and the kitchen selves. Instructions and data that are most likely to be used are stored in the cache memory. This makes execution process to finish earlier compared to having to look for them in the main memory. Besides, since accessing cache is faster than the main memory, the time gained can be spent on processing the next instructions and data.

2.2.2 CPU Caches

Before CPU cache memory was introduced, the CPU was directly connected to the main memory, however, that changed soon. A CPU cache is a cache memory placed between the processor[1] and the main memory. The CPU first looks for data in cache which is 10 to 20 times faster than accessing the main memory [1].

Initial CPU Configuration

When cache memory made its initial entrance in computers, the CPU core was directly connected to the cache. This was in turn connected through the system bus to the main memory [12]. The CPU had no direct communication with the main memory and all *loads* and *stores* had to go through the cache [12]. Figure 2.1 illustrates the concept of this simple one level configuration.

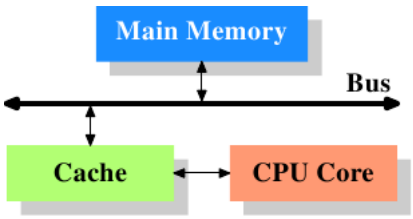

Figure 2.1: Simple Cache Configuration[12]

CPU Multi Level Caching

In the early 90's Intel decided to separate L1 cache to two caches. One for code instructions and one for data. Results showed that this configuration would produce better results. Based on this distinction, *I1* (or L1i) will refer from now on to the level 1 instruction cache, while *D1* (or L1d) to the level 1 data cache [12]. Caches containing both instructions and data are called *unified* caches.

Moreover, modern computers use multiple levels of cache. This served as a solution to the increased speed difference between the cache memory and the main memory, which increased to a point that more levels of cache had to be added, slower but bigger than the first level cache. Figure 2.2 illustrates both L1 cache separation and multi-level caching.

[1]The terms 'CPU' and 'processor' are used interchangeably.

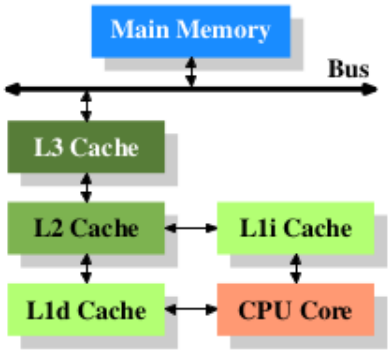

Figure 2.2: Multiple Level Cache Configuration[12]

Computers have up to 3 levels of cache whose typical sizes and access times can be seen in Table 2.1.

Cache Level	Typical Size	Typical Access Time
1	16KB - 64KB	5 - 10 ns
2	128KB - 4MB	40 - 60 ns
3	4MB - 8MB	60 - 100 ns

Table 2.1: Typical Cache Information[1]

2.2.3 How CPU Caches Work in More Detail

Associativity

Associativity[2] works as described below. Multiple *words* (fixed size groups of bits) are called *blocks* or *lines*. Each block contains a *tag* to indicate the location of the memory words were copied from. Associativity describes where in cache, blocks (or lines) are placed. The most used scheme is called *set associative*. A group of blocks in the cache is called a *set*. To place a block in a set, a block has to be mapped onto a set and then the block can be placed anywhere in that set. To find a block, the block address has to be mapped with the set and then search the set to find the block. The search operation is usually done in parallel. The set is chosen by the following operation:

$$(Block\ Address)\ MOD\ (Number\ of\ Sets\ in\ Cache)$$

A cache placement is called N-way associative if N sets exist in the cache. The extreme cases of set associativity are named *direct-map* cache and *fully associative* cache [14]. Direct-mapped caches have one block per set, thus "a block is always placed in the same location"[14]. Fully-associative caches have only one set, thus "a block can be placed anywhere"[14]. Figure 2.3 illustrates cache associativity.

[2]Associativity is also present on TLB, which is described later. It is not restricted to CPU caches.

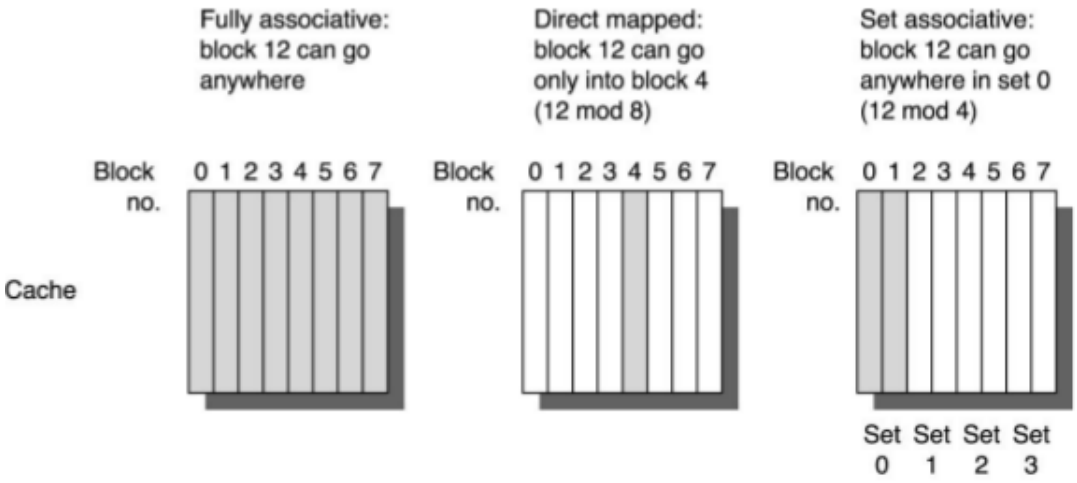

Figure 2.3: Cache Associativity[14]

CPU Cache Entries

Line size varies from 32 to 256 bytes [1] while the norm is 32 to 64 bytes [19]. A cache entry is created when a line is being copied from main memory to cache. A cache entry consists of the copied data and the location of the memory data were copied from (also called Tag) [14]. When a CPU has to read or write a location in main memory, it looks in the cache lines for that memory location. "In case of a cache hit, the CPU reads or writes the data in the cache line" [14], whereas in case of a cache miss, a new entry is created and data are copied from memory [14].

CPU Cache Entry Structure

The structure of a cache entry is the following:

$$Tag \mid Index \mid Offset$$

where:
Tag: Location of the memory data were copied from.
Index: Cache line in which the data exist.
Offset: Data within the stored data block within the cache line.

Cache Miss Entry Replacement

In case of a cache miss, new data must be entered in the cache. If the cache is full, then previous entries must be deleted. Algorithms that help to complete this procedure are briefly described below[3]:

Random Replacement (RR)
Randomly selects an entry to be removed.

Least Recently Used (LRU)
Removes the least recently used entries.

Least Frequently Used (LFU)
Entry usage is counted and the one used the least is removed.

First Entry Removal (FER)
Removes the first entry entered in the cache.

[3]The same techniques exist for TLB Entry Replacements. They are not limited to CPU caches.

Last Entry Removal (LER)
Removes the last entry entered in the cache.

2.2.4 Translation Look-Aside Buffer (TLB)

What is TLB, TLB hit & miss

Besides CPU caches, a second important factor that increases a program's execution speed is the TLB. The TLB is a hardware cache of popular virtual-to-physical address translations and is part of the memory-management's unit (MMU) [3]. Whenever a virtual memory reference occurs, the hardware first looks the TLB for a corresponding physical address. If a translation is held there (*TLB hit*) the translation occurs quickly, without having to look into the page table where all translations are being held. Otherwise, in case of a *TLB miss*, the page table is used for the translation [3].

How Does TLB Work in More Detail

"Programs generate virtual addresses (VA). Each VA is passed from CPU to MMU. The MMU translates the VA to a physical address (PA) and passes the PA to memory" [13]. If there was no caching, then three memory accesses would be needed and "each datum would need four accesses to memory" [13]. If that was the case, virtual memory (VM) access would be four times slower than a physical access [17] and that would not be practical. "A trick removes most of this performance penalty: modern CPUs use a small associative memory to cache the page table entries (PTEs) of recently accessed virtual pages. That is the TLB" [17].

"The TLB works as follows. On a virtual memory access, the CPU searches the TLB for the virtual page number of the page that is being accessed, an operation known as TLB lookup. If a TLB entry is found with a matching virtual page number (VPN), a TLB hit occurrs and the CPU can go ahead and use the PTE stored in the TLB entry to calculate the target physical address. The reason why the TLB makes virtual memory practical is because it is small—typically on the order of a few dozen entries—it can be built directly into the CPU and runs at full CPU speed. This means that as long as a translation can be found in the TLB, a virtual access executes just as fast as a physical access. Indeed, modern CPUs often execute faster in virtual memory, because the TLB entries indicate whether it is safe to access memory speculatively (e.g. to prefetch instructions)" [17].

Handling TLB Misses

A TLB miss can be handled both by hardware and software. "In practice, with care there can be little performance difference between the two approaches, because the basic operations are the same in either case" [23].

In case of CISC architecture, the hardware would handle the TLB miss entirely [3]. "To do this the hardware has to know exactly where the page tables are located in memory, as well as their exact format. In case of a miss, the hardware would walk the page table, find the correct page table entry and extract the desired translation, update the TLB with the translation and retry the instruction. Intel's x86 architecture has hardware managed TLB" [3].

In case of a RISC architecture, or more modern than RISC, the TLB is managed by the software [3]. "On a TLB miss, the hardware simply raises an exception which pauses the current instruction stream, raises the privilege level to kernel mode and jumps to trap handler. Trap handler is code within the OS that is written with the express purpose of handling TLB misses. When run, the code will lookup the translation in the page table, use special 'privileged' instructions to update the TLB, and return from the trap; at this point, the hardware retries the instruction (resulting in a TLB hit)" [3].

Typical TLB

Typical TLB values may be seen in Table 2.2.

TLB Size	16–512 entries
Block Size	1–2 page table entries (typically 4–8 bytes each)
Hit Time	0.5–1 clock cycle
Miss Penalty	10–100 clock cycles
Miss Rate	0.01%–1%

Table 2.2: Typical TLB Information[23]

TLB Entries

TLB entries refer to the form of data inside the TLB caches. "A TLB might have 32, 64 or 128 entries and be what is called fully associative. This just means that any given translation can be anywhere in the TLB, and that the hardware will search the entire TLB in parallel to find the desired translation" [3]. A typical TLB entry might look like this:

$$VPN \mid PFN \mid Other\ Bits$$

where:
VPN: Virtual Page Number
PFN: Page Frame Number
VPN and PFN are present in all entries, since a translation could end up in any of these locations. "The hardware searches the entries in parallel to see if there is a match"[3]. Other bits refer to bits that help the translation process. Among others, they may include a valid bit, which indicates whether the entry has a valid translation or not, a protection bit, which indicates how a page can be accessed, and an address space identifier (ASID), which indicates the process by which a translation can be used [3].

Multiple Level TLBs

Nowadays, multi-level TLB caches exist in the same fashion as for CPU caches. The higher the level of the TLB, the larger and slower it is [12]. L1 and L2 are common at the time of writing. L1 TLBs are often separated to be used for instructions (ITLB) and data (DTLB) and exist per processor, while L2 TLBs are unified and are shared among the processors (STLB[4]).

How does the TLB Fit in the Bigger Picture

TLB is not placed in the same way in all architectures. Assuming a modern Intel x86 architecture, the TLB is placed between the CPU and the CPU caches described earlier. This scheme allows for virtual addresses to be translated before being entered in the CPU caches. Intel's i7 processor cache structure can be seen in figure 2.4. ITLB communicates directly with both STLB and level 1 instruction cache (I1), while DTLB communicates with STLB and the level 1 data cache (D1). Figure 2.4 was taken by Intel's documentation ([6]), although, since it contained an amount of information that might confused the reader, it was edited to show only CPU cache and TLB relevant information.

[4]Note that for the rest of the project the term L2TLB is likely to be used more often, than STLB, in order to refer to the second level TLB.

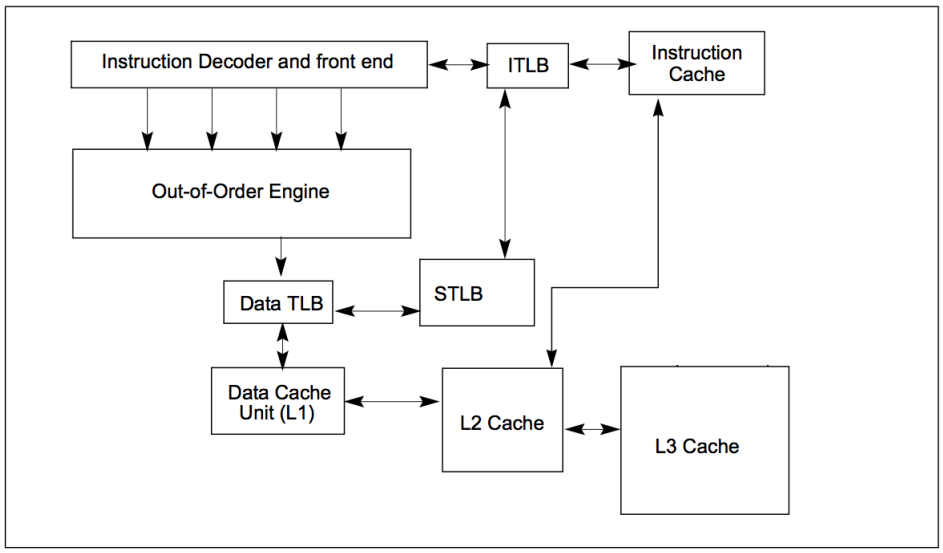

Figure 2.4: How does TLB fit the bigger picture [6]

2.3 Valgrind Suite & Cachegrind

2.3.1 Introducing Valgrind Suite & Cachegrind

Valgrind is a "system for debugging and profiling Linux programs. With Valgrind's tool Suite you can automatically detect many memory management and threading bugs, avoiding hours of frustrating bug-hunting, making your programs more stable. You can also perform detailed profiling to help speed up your programs" [9].

Cachegrind is a cache and branch-prediction[5] profiler that belongs to the Valgrind Suite. It performs detailed simulation of the I1, D1 and LL caches in your CPU so it can identify the cache misses in the source code .

2.3.2 Running Cachegrind

To run Cachegrind on a program prog, run:
`valgrind --tool=Cachegrind prog`
"The program will execute (slowly). Upon completion, summary statistics that look like Figure 2.5 will be printed"[11]. "Cache accesses for instruction fetches are summarised first, giving the number of fetches made (this is the number of instructions executed, which can be useful to know in its own right), the number of I1 misses, and the number of LL instruction (LLi) misses. Cache accesses for data follow. The information is similar to that of the instruction fetches, except that the values are also shown split between reads and writes (note each row's rd (read) and wr (write) values add up to the row's total). Combined instruction and data figures for the LL cache follow that. Note that the LL miss rate is computed relative to the total number of memory accesses, not the number of L1 misses" [11].
i.e.

$$(ILmr + DLmr + DLmw)/(Ir + Dr + Dw)$$

not

$$(ILmr + DLmr + DLmw)/(I1mr + D1mr + D1mw)$$

[5]Branch prediction is beyond the scope of this project and, thus, it will not be focused.

```
==31751== I   refs:      27,742,716
==31751== I1  misses:           276
==31751== LLi misses:           275
==31751== I1  miss rate:        0.0%
==31751== LLi miss rate:        0.0%
==31751==
==31751== D   refs:      15,430,290  (10,955,517 rd + 4,474,773 wr)
==31751== D1  misses:        41,185  (    21,905 rd +    19,280 wr)
==31751== LLd misses:        23,085  (     3,987 rd +    19,098 wr)
==31751== D1  miss rate:        0.2% (       0.1%   +       0.4%)
==31751== LLd miss rate:        0.1% (       0.0%   +       0.4%)
==31751==
==31751== LL misses:         23,360  (     4,262 rd +    19,098 wr)
==31751== LL miss rate:         0.0% (       0.0%   +       0.4%)
```

Figure 2.5: Cachegrind Output [11]

It should be noted that Cachegrind when run with multithreaded programs, it serialises all threads and analyses the sequential code flow [20].

2.3.3 Why Use Cachegrind

As it has already been stated, the use of caches may have a crucial effect on program speed. Some popular techniques exist to optimise cache use of array-based programs, however, the situation on general purpose programs remains vague [21]. Cachegrind provides detailed information upon the use of caches so that the programmer can make changes to the code resulting in a more efficient and effective use and, thus, increase program speed and make better use of the provided hardware.

Moreover, most profiling tools provide only global statistics, such as the number of cache hits and misses, which are of limited help to the programmer. Cachegrind provides cache information tied to particular parts of a program [21]. This provides the programmer with more information and a bigger picture to reason about.

2.3.4 Why Cachegrind Limits Results to L1 and LL

The reason that Cachegrind limits its results to L1 and LL and does not already include L2 in case of 3 levels of cache is because "the last-level cache has the most influence on runtime, as it masks accesses to main memory. Furthermore, the L1 caches often have low associativity, so simulating them can detect cases where the code interacts badly with this cache (eg. traversing a matrix column-wise with the row length being a power of 2)" [10].

2.4 Previous Work

Section 2.4.1 explains the various existing profiling methods and their categories, while section 2.4.2 focuses on the profiling method used by Cachegrind and describes it.

Sections 2.4.3, 2.4.4, 2.4.5 and 2.4.6 present, respectively, a brief documentation of the files consisting Cachegrind, an explanation of how does Cachegrind communicate with Valgrind, the main data structures used and, finally, CPUID instructions.

2.4.1 Profiling Methods

This section draws upon approaches used for cache profiling. Cache profiling methods can be split into two major categories: hardware based and software based.

Hardware Performance Counter Based

Modern processors have a set of registers to count CPU events that are called performance counters. Hardware-based cache profiling, in general, is efficient and cheap, since memory accesses are done by the hardware and then only performance registers need to be read to acquire those values [19]. This method also, "allows for improvement on the same hardware, while an improvement obtained by software simulation is not necessarily effective on the actual hardware" [19]. "But, at the same time this profiling method is naturally bound to the cache model of the underlying hardware. Consequently, hardware performance counter based methods are best suited for the practical case that a user wants to speed up a specific program on a specific architecture" [19].

Software Simulation Based

This type of simulation is mostly preferred by those who intend to study cache performance in general, than improve specific software on specific hardware. Software simulation allows for "extreme" analysis that cannot be done by hardware methods, "such as changing cache parameters, and experimenting with new cache architectures" [19]. Software simulation can be split further in two categories: trace-driven and execution-driven simulation.

Trace-driven simulation (Cachegrind belongs in this category [21]), simulates a portion of the hardware, such as caches and memory systems. For simulation to happen the trace of a program is required, since it contains the history of memory accesses. "Trace-driven simulations are further divided into two types according to the way they collect traces: object-level instrumentation and source-level instrumentation" [19]. Source-level instrumentation involves altering the source code, while, object-level instrumentation occurs at the level of machine code. Figure 2.6 illustrates this concept.

Execution-driven simulation "simulates CPU instructions and usually several I/O devices and executes a program on a virtual CPU, by interpreting every instruction" [19].

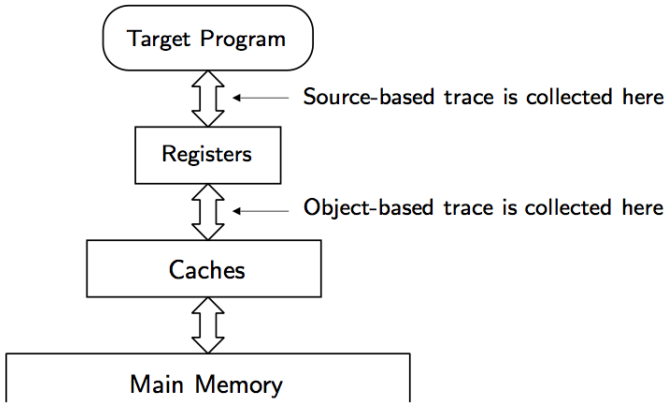

Figure 2.6: Difference between source-level and object-level instrumentation [19]

2.4.2 Dynamic Binary Instrumentation

Valgrind is a dynamic binary instrumentation (DBI) framework. To link with the above categories, it belongs in the trace-driven software simulation category and performs object-level instrumentation. Instrumentation refers to monitoring or measuring the level of a prod-

uct's performance, diagnosing errors and writing trace information [8]. Instrumentation is achieved by adding extra code to a program [21].

The term "*binary*" refers to analysis of a program "at the level of machine code, stored either as object code (pre-linking) or executable code (post-linking). Binary analysis includes analyses performed at the level of executable intermediate representations" [21]. Binary analysis is the feature that makes Valgrind a language independent tool. On the other hand, the term "*dynamic*" refers to analysis that occurs during run-time.

There are two fundamental ways for a DBI framework to represent code and allow instrumentation, *Disassemble & Resynthesize* (D&R) and *Copy & Annotate* (C&A) [22]. Valgrind uses D&R, "where the machine code is converted to an intermediate representation (IR) in which each instruction becomes one or more IR operations. This IR is instrumented (by adding more IR) and then converted back to machine code" [22].

As it has already been stated, Cachegrind is a tool built on top of the Valgrind framework, specialising in cache profiling. The following sections, 2.4.3 to 2.4.6, explain Cachegrind in more depth in order to gain a better understanding.

2.4.3 Cachegrind's File Structure

Cachegrind mainly consists of files written in C. Many other types of files exist, but they will not be described in this section, since they are not of any importance to L2 cache inclusion and TLB measuring, which is the aim of this project. Furthermore, the files consisting the core of Cachegrind (C files) can be split into two categories: a) files that perform cache simulation & profiling and b) files that contain architecture specific definitions.

a) Cache Simulation & Profiling Files

Cache Simulation & Profiling Files	
File Name	Description
cg_main.c	Contains everything but the simulation itself.
cg_sim.c	Contains the cache simulation.
cg_merge.c	Contains a program that merges multiple Cachegrind output files. "cg_merge can be used as an optional intermediate step to sum together the outputs of multiple Cachegrind runs into a single file which you then can use an an input for cg_annotate" [11]
cg_branchpred.c	Contains the actual branch predictor simulator. Can be used as an optional tool.
cg_arch.h	Contains cache configuration definitions.
cg-arch.c	Contains the cache configuration.

b) Architecture Specific Definitions

Architecture Specific Definitions	
File Name	Description
cg-arm.c	ARM specific definitions.
cg-mips32.c	MIPS specific definitions.
cg-ppc32.c	PPC32 specific definitions.
cg-s390x.c	s390x specific definitions.
cg-x86-amd64.c	Intel x86 and AMD64 specific definitions.

2.4.4 Cachegrind's Link with Valgrind

As it already has been stated, Cachegrind is a tool within the Valgrind Suite and as such it must contain 4 functions in its main file (cg_main.c), whose usual names are the following:

- `pre_clo_init()`

- `post_clo_init()`
- `instrument()`
- `fini()`

In the case of Cachegrind the above functions contain the prefix "cg_" (short for Cachegrind), i.e. cg_pre_clo_init(), cg_post_clo_init(), cg_instrument(), and cg_fini(). What each of these 4 functions should contain can be seen below.

pre_clo_init()
pre_clo_init() is responsible for most of the initialisation. Various details such as program name, version, description, author and more can be set here. Moreover, various "needs" can be set for the tool such as: record, report and suppress errors; process command line options; wrap system calls; and more. Finally, "the tool can indicate which events in core it wants to be notified about" [11].

post_clo_init ()
post_clo_init() is used only if a tool provides command line options ('clo') and initialisation is required after option processing has taken place [11].

instrument()
instrument() allows the tool developer to instrument VEX IR, "which is Valgrind's RISC-like intermediate language" [11]. It is the most important function among the 4 described, since it contains a tool's core code and does most of the work the tool has to do. fini() (described right after) presents results based on data gathered by instrument() [11].

fini()
This is where final results are presented as a summary of the information collected in the previous functions. Moreover, if the program should generate any log files, they should be written here [11].

2.4.5 Cachegrind's Main Data Structures

Cachegrind's most important data structures have been initially noted and explained by Nicholas Nethercote, the creator of Valgrind and Cachegrind, in his PhD thesis in [**nnpdh**]. Since then, they have been revised by Valgrind developers. This section presents a brief explanation of Cachegrind's main data structures, i.e. the Cost Centre table, the Instruction Info table, the String table and cache_t.

Cost Centre Table
The Cost Centre table contains the "per-line cost centres (CCs)" [21]. A CC table data structure is capable of the following [20]:

- Holds the per-source-line hit/miss stats, grouped by file/function/line.
- Is an ordered set of CCs. CC indexing done by file/function/line.
- Is traversed for dumping stats at end in file/function/line hierarchy.

The Cost Centre table is defined as follows:

```
static OSet* CC_table;
```

The code of three important Cost Centre structures can be seen below:

```
1 typedef struct {
2       ULong a;  /* total # memory accesses of this kind */
3       ULong m1; /* misses in the first level cache */
4       ULong mL; /* misses in the last level cache */
5 }CacheCC;
```

```
typedef struct {
      ULong b;  /* total # branches of this kind */
      ULong mp; /* number of branches mispredicted */
}BranchCC;
```

```
typedef struct {
   CodeLoc  loc; /* Source location that these counts pertain to */
   CacheCC  Ir;  /* Insn read counts */
   CacheCC  Dr;  /* Data read counts */
   CacheCC  Dw;  /* Data write/modify counts */
   BranchCC Bc;  /* Conditional branch counts */
   BranchCC Bi;  /* Indirect branch counts */
} LineCC;
```

Instruction Info table

The Instruction Info table holds the cached information about each instruction that is used for simulation. The Instruction Info table is defined as follows:

```
static OSet* instrInfoTable;
```

"Each instruction instrumented is given a node in the table; each node has the type _InstrInfo" [21] which can be seen below:

```
typedef struct _InstrInfo InstrInfo;
struct _InstrInfo {
   Addr    instr_addr; //instruction address
   UChar   instr_len;  //instruction length
   LineCC* parent;     // parent line-CC
};
```

String Table

The String table is considered to be a secondary data structure while the above are primary data structures, but it is worth explaining. The string table is capable of the following [20]:

- Holds strings, avoiding dups.
- Can be used for filenames and function names, each of which will be pointed to by one or more CCs.
- Allows equality checks just by pointer comparison, which is good when printing the output file at the end.

The String table is defined as follows:

```
static OSet* stringTable;
```

cache_t

cache_t is a structure used for cache simulation which contains the size of the cache and the line size in bytes as well as the associativity of the cache.

```
typedef struct {
   Int size;       // bytes
   Int assoc;
   Int line_size;  // bytes
} cache_t;
```

2.4.6 CPUID

CPUID is an instruction used to retrieve information about a CPU. This could be the processor signature, vendor name, model number and information about the features supported and implemented by the processor [7].

Cachegrind makes use of the CPUID instruction in the architecture specific definition files (see section "*2.4.3 b) - Architecture Specific Definitions*"), in order to retrieve information about caches, specifically to get L1 and LL cache configuration descriptors i.e. their size, line size and associativity.

2.5 Technology

In this section the exact technological resources to extend and build Cachegrind are described. These include the computer architecture where extensions will take place, the compiler that will be used, as well as the instructions on how to install Valgrind and how to compile any changes in Cachegrind.

2.5.1 Computer

The computer architecture used is the x86 architecture, given that the initial goal is to make any changes work for x86 architecture. When the changes are completed, we could try to implement them to other architectures and test them using a simulator, since it could be difficult to get access to machines of ppc, s390x and ARM architectures.

2.5.2 Compiler

The compiler used to compile Cachegrind is set in Cachegrind's Makefile and is gcc. The current version of gcc, in the computer in use, as shown by `gcc --version` is `v.4.6.3.`

2.5.3 Install Valgrind Suite from source code

To install Valgrind Suite from source, one has to extract Valgrind tar.bz2 file, then enter in the extracted direcotry (`valgrind-3.8.1` in the case of this project) and execute the following commands on terminal:

```
./configure
make
sudo make install
```

Verify that it works by executing the following without obtaining any errors:

```
valgrind ls -l
```

2.5.4 Compile & Install Cachegrind Changes

If changes in source code take place, one can compile and install them just by compiling the Cachegrind tool instead of all the Valgrind suite. Thus, given that Valgrind is already installed, in order to compile Cachegrind and install changes, one has to go to Cachegrind's directory (`valgrind-3.8.1/cachegrind`) and type:

```
make
sudo make install
```

2.5.5 Run Cachegrind Without Installation

Since one might want to run Cachegrind without installing it, or might not have the right permissions to install it, Valgrind provides an alternative way to run its tools without installing anything, i.e. avoiding anything mentioned in the last two subsections. Given that one is within `valgrind-3.8.1`, may execute the following:
`./configure`
set `VALGRIND_LIB` environment variable:
`export VALGRIND_LIB=$PWD/.in_place`
`make`
then change directory to `coregrind`:
`cd coregrind`
and execute Cachegrind as follows:
`./valgrind --tool=cachegrind <path_to_file>`

2.6 Testing Strategy

This section explains the testing and evaluation approach that will be followed for L2 cache inclusion and TLB measuring.

Testing in both L2 cache inclusion and TLB measuring will take place by executing programs whose behaviour can be predicted and then, results produced will be checked on whether they are along the lines of the expected ones or not. Instead of a check, a mathematical proof could have been devised, however, since the number of factors that might affect the result are very difficult to be determined, it was decided not to proceed with it.

Chapter 3

L2 Cache Inclusion

3.1 Introduction

Cachegrind, version 3.8.1, only provides information for L1 (level 1) data and instruction caches, as well as for the LL (last level) cache. In the common case that three levels of cache exist, Cachegrind limits its results to L1 (for each of data and instructions) and L3, neglecting L2.

This chapter focuses on L2 cache and describes the way Cachegrind was approached and extended to provide information for L2 cache when three levels are present. It starts by explaining the framework within which L2 cache extension took place and the necessary changes made so that other tools would not be affected. It, then, explains how Cachegrind works, by shedding light upon its structure, as well as on how it was extended to include L2 cache. All of the sections focus on providing high-level explanations. In some cases, however, low-level explanations, i.e. C code, are provided to ease understanding; they are, though, limited.

Moreover, the extended version of Cachegrind is run and the results are presented and discussed. A check of the validity of the results follows and, finally, a comparison of the execution times between the initial version and the extended one is performed aiming to observe the effect of the extensions on execution speed.

It is important to note that this project is only concerned to make any extensions work in an Intel x86 machine and should be judged upon that. As a result, it describes the procedures followed for an Intel x86 extension. However, if there is time, attempts will take place to make extensions work on AMD and other architectures.

It should also be noted that whenever a function call is encountered whose parameters are of no importance, dots will be used to represent them. i.e. given a function foo(int n1, int n2, ... , int nn), if parameters do not matter, it will be referred to by foo(..).

3.1.1 General approach to L2 cache extension

As it has already been stated, at the moment, Cachegrind provides information for L1 and LL caches only. Since L1 caches are separated into instruction and data caches, the mechanism used to simulate those two cannot easily be used to simulate caches that are not separate for instructions and data, and thus, cannot be used to simulate the Intel x86 L2 cache. However, Cachegrind also provides information for the LL cache which, depending on the total number of caches, can represent either L2 or L3. Thus, the mechanism to simulate and provide information for the L2 cache is already implemented within Cachegrind and does not have to be created from scratch. What is needed, are some extensions to include L2 cache results when three levels of cache exist, i.e. not to neglect L2. Thus, explicit L2 cache inclusion should be treated in a very similar manner that Cachegrind already treats the LL cache. An analytic explanation of how L2 cache inclusion was done can be seen in sections 3.2 and 3.3. These sections describe the data structures used and the extensions made.

3.1.2 Isolate changes to Cachegrind

Before proceeding to any changes, we had to ensure that they would not cause any inconveniences to other Valgrind tools. *Callgrind* is the only tool (apart from Cachegrind) that makes use of libraries defined within Cachegrind. Specifically, it makes use of cg_arch.h, which is a file whose code was changed later, so Callgrind would not be able to use it. As a result, a copy of that file was made and renamed to cg_arch_init.h. In addition, Callgrind's simulation file (sim.c) was changed to include the correct header (cg_arch_init.h). Without that change, Callgrind would not be able to compile.

3.2 How does Cachegrind work

This section explains in more depth the mechanics beneath Cachegrind. As described in section "*2.4.4 — Cachegrind's Link with Valgrind*", Cachegrind, and every Valgrind tool, must contain four functions: `pre_clo_init()`, `post_clo_init()`, `instrument()` and `fini()`. In this section we describe how these functions work and the notions underlining them. Moreover, we provide an in-depth explanation of Cachegrind's data structures and depict its overall structure. The concepts explained here are crucial in understanding how L2 cache was incorporated in the current Cachegrind structure.

3.2.1 Overall Structure

The fact that Valgrind requires its tools to use the 4 functions stated above, makes it easier to divide the concept underlining the tool into 4 parts. This abstraction helps us understand how the tool works and, thus, be in a better position to extend and explain it. The notion behind any Valgrind tool is to use Valgrind's core in order to get notified for various events taking place in the system and, by creating tool-specific variables, structures and functions, be able to profile and measure different parts of the system. Cachegrind traces all memory events that take place and depending on the kind of the event (such as instruction reads, data reads, data writes, etc.) gets updated and, thus, is able to simulate the caches.

The first function called is `cg_pre_clo_init`, which limits itself to some initialisations for different aspects of Cachegrind. The second function called is `cg_post_clo_init` which, if there are existing arguments, processes them and makes any necessary initialisations based on the arguments. Then `cg_instrument` is called where instrumentation takes place, a fact that turns this function into the most important function of any tool. "Instrumentation is a technique for inserting extra code into an application to observe its behaviour" [16]. In the case of Cachegrind, cg_instrument traces memory events and, based on their kind, it calls the function that should process them. That function (the one just called, not cg_instrument) updates some variables and calls some "helper" functions which are entirely written by the developer[1] of the tool and they act upon tool-specific variables the developer has created. For Cachegrind, these are variables that allow cache simulation to take place. Finally, the last function called is `cg_fini`, which presents results based on data gathered during the instrumentation phase. For Cachegrind, that is an analysis of data gathered for the various cache levels.

A more in-depth analysis of what happens on every of these 4 function calls, generally and in relation to Cachegrind, can be seen in the following sections 3.2.3 — 3.2.6. However, before the 4 function analysis takes place, it is important to explain the data structures, so that they can be referenced while analysing the 4 functions.

3.2.2 Data Structures

At this point, the reader is encouraged to recall information covered in section "*2.4.5 — Cachegrind's Main Data Structures*". The present section describes in more depth the

[1] The two functions called by cg_instrument so far are written by the tool developer, but the code used is standardised and is used by several tools, thus, it cannot be argued that they have been entirely written by the tool developer.

data structures described in that section and adds a few more to provide a bigger and more complete picture of Cachegrind.

OSet

Although OSet is a data structure creator housed in the Valgrind core and is used by many tools rather than being a Cachegrind-specific data structure, it is worth mentioning, since some of the Cachegrind-specific data structures that will be discussed use it.

OSet (Ordered Set) is a data structure with fast insertion, lookup and deletion of elements, with no duplicates allowed [20]. OSet has two interfaces: OSetGen_ and OSetWord_. The former provides a totally generic interface, which allows any kind of structure to be put into the set, while the later provides an easier-to-use interface, but is limited in terms of the structures that can be put in the set [20]. This dissertation will only tackle OSetGen_. OSetGen_ provides a few functions to create (`VG_(OSetGen_Create)`), destroy (`VG_(OSetGen_Destroy)`), insert, remove and lookup elements, as well as to allocate and free nodes (`VG_(OSetGen_AllocNode)`, `VG_(OSetGen_FreeNode)`). OSet and the collection of functions provided, i.e. the interface, can be used by tools as a database to store data.

Cost Centre Table

The Cost Centre table is a Cachegrind specific structure that contains the "per-line cost-centres (CCs). Every source line that gets instrumented and executed is allocated its own cost centre in the table, which records the number of cache accesses and misses caused by the instructions derived from that line" [21]. It also groups the per-source-line hit/miss stats by file/function/line [20] with the aid of `CodeLoc` (Code Location) structure. The cost centre table is defined as follows:
`static OSet* CC_table;`
and created using `VG_(OSetGen_Create)` with chosen structure to be put in the set `lineCC`:
`CC_table =VG_(OSetGen_Create) (.., lineCC, ..);`

`lineCC` represents every source line instrumented and consists of elements of 2 different cost centre structures: `CacheCC` and `BranchCC` as well as one element of `CodeLoc` to hold the file, function and line information. `lineCC` structure is shown in the following code snippet with comments explaining the purpose of each variable.

Listing 3.1: LineCC

```
1  typedef struct {
2     CodeLoc  loc; /* Source location that these counts pertain to ↩
          */
3
4     CacheCC  Ir;  /* Instruction read counts */
5     CacheCC  Dr;  /* Data read counts */
6     CacheCC  Dw;  /* Data write/modify counts */
7
8     BranchCC Bc;  /* Conditional branch counts */
9     BranchCC Bi;  /* Indirect branch counts */
10 } LineCC;
```

An analysis of each of CodeLoc, CacheCC and BranchCC follows:

- **CodeLoc**

  CodeLoc holds the file name, the function name and the line number of every source line instrumented. Thus, CodeLoc contains 3 elements: **file**, **fn** and **line**.

  The following code snippet shows how is CodeLoc represented in terms of C code:

Listing 3.2: CodeLoc

```
1  typedef struct {
```

```
2     Char* file;
3     Char* fn;
4     Int   line;
5  }
6  CodeLoc;
```

- **CacheCC**

  CacheCC is used for cache profiling and contains 3 elements: **a**, **m1** and **mL** that hold the number of accesses, L1 misses and LL misses respectively [21]. The following code snippet demonstrates how is CacheCC represented in terms of code.

Listing 3.3: CacheCC

```
1  typedef
2     struct {
3        ULong a;  /* total # memory accesses of this kind */
4        ULong m1; /* misses in the first level cache */
5        ULong mL; /* misses in the last level cache */
6     }
7     CacheCC;
```

- **BranchCC**

  BranchCC is used for branch prediction, which is beyond the scope of this project; it is explained, however, in order to provide the reader a bigger picture of the cost centre structures.

  BranchCC contains 2 elements: **b** and **mp** that hold the number of branches and the mispredicted branches respectively. The following code snippet represents how is BranchCC represented in terms of code.

Listing 3.4: BranchCC

```
1  typedef
2     struct {
3        ULong b;  /* total # branches of this kind */
4        ULong mp; /* number of branches mispredicted */
5     }
6     BranchCC;
```

Figure 3.1 visualises the concepts described above and the connections between them.

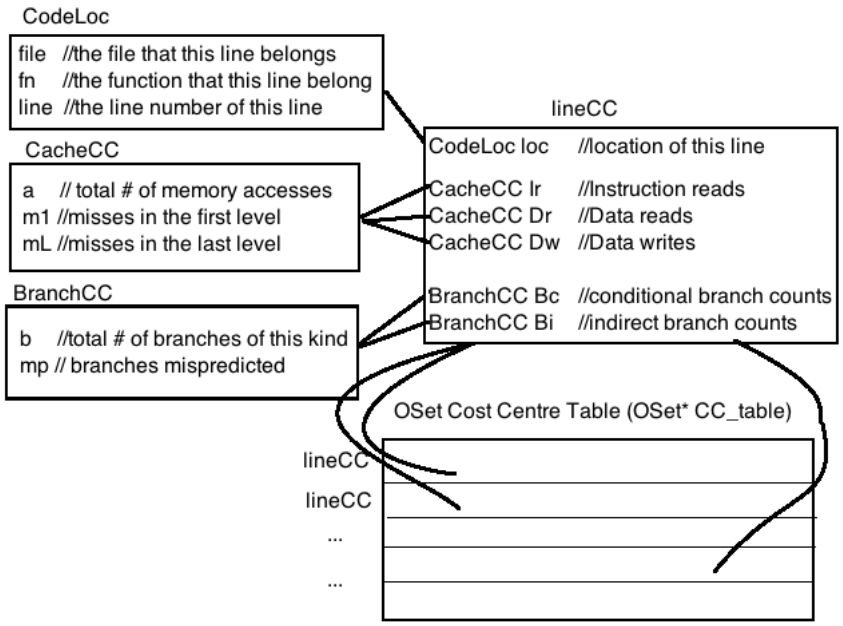

Figure 3.1: Cost Centre Structures

InstrInfo Table

The InstrInfo (Instruction Information) table holds cached information for each instruction that is used for simulation. The code is broken into small blocks called *superblocks*, where each superblock represents from 1 to 50 instructions. Each superblock (type **SB_info**) has some information stored for it, such as its address, the number of instructions contained, and information for each of these instructions (type **InstrInfo**) [20].

The following 2 code snippets represent SB_info and InstrInfo in terms of C code respectively:

Listing 3.5: SB_info

```
1 typedef struct _SB_info SB_info;
2 struct _SB_info {
3    Addr      SB_addr;      // key;  MUST BE FIRST
4    Int       n_instrs;
5    InstrInfo instrs[0];
6 };
```

Listing 3.6: InstrInfo

```
1 typedef struct _InstrInfo InstrInfo;
2 struct _InstrInfo {
3    Addr    instr_addr;
4    UChar   instr_len;
5    LineCC* parent;         // parent line-CC
6 };
```

For each superblock (SB_info), each InstrInfo in the list holds information about the instruction (instruction length and instruction address) plus a pointer to its lineCC [20]. Finally, the InstInfo table is defined as follows:
`static OSet* instrInfoTable;`

Created using `VG_(OSetGen_Create)` with SB_info as the chosen structure to be put into:
`instrInfoTable = VG_(OSetGen_Create)(.., SB_info, ..);`
Figure 3.2 visualises the above concepts and the relationship between them.

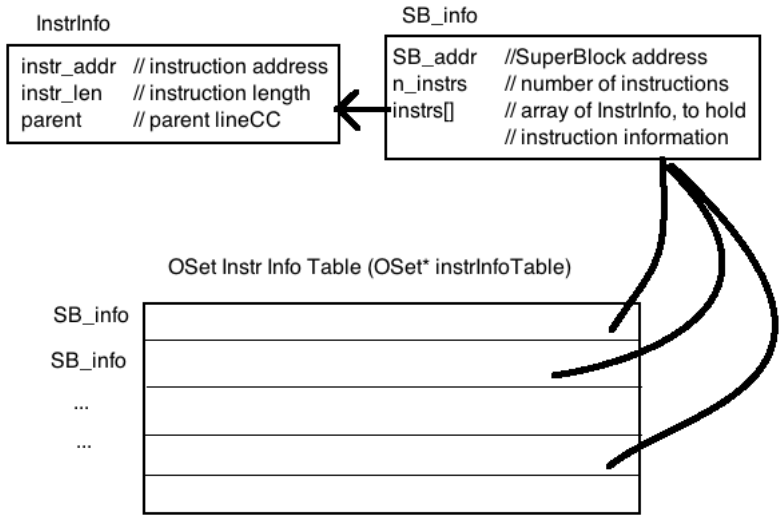

Figure 3.2: InstrInfo Table

String Table

The String Table is used to hold strings of filenames and function names mainly used by CodeLoc. It, also, allows for equality checks between strings just by pointer comparison. The string table is defined as follows:
`static OSet* stringTable;`
Created using `VG_(OSetGen_Create)` with `Char*` as the chosen structure to be put into:
`stringTable = VG_(OSetGen_Create)(.., Char*, ..)`

cache_t

cache_t is a structure used for cache simulation and is used to hold basic cache technical characteristics, such as the cache size, cache associativity and cache line size. Cache size and cache line size are measured in bytes.
The following code snippet shows how cache_t is represented in terms of C code:

Listing 3.7: cache_t

```
1 // For cache simulation
2 typedef struct {
3    Int size;       // bytes
4    Int assoc;
5    Int line_size;  // bytes
6 } cache_t;
```

cache_t2

cache_t2 is used for cache simulation and contains a number of details for the cache it represents. These are the cache size (**size**), associativity (**assoc**), line size (**line_size**), number of sets (**sets**), number of sets minus 1 (**sets_min_1**), line size in bits (**line_size_bits**) which is calculated by log2(line_size), **tag_shift** which is equal to line_size_bits + log2(sets),

desc_line which contains a text of the particular cache configuration in one of the two following forms: "(size)B, (line_size)B, (assoc)-way associative" or "(size)B, (line_size)B, direct-mapped" and, finally, a pointer named **tags** pointing to an array.
The following code snippet shows how is `cache_t2` represented in terms of C code:

Listing 3.8: cache_t2

```
1  typedef struct {
2     Int           size;             /* bytes */
3     Int           assoc;
4     Int           line_size;        /* bytes */
5     Int           sets;
6     Int           sets_min_1;
7     Int           line_size_bits;
8     Int           tag_shift;
9     Char          desc_line[128];
10    UWord*        tags;
11 } cache_t2;
```

Since cache_t2 holds all cache_t information, raises the question why both data structures exist. The answer to this is based on two reasons: First, because Cachegrind makes a clear distinction between the simulation phase and the rest of the tool. cache_t is used before simulation takes place, in order to hold basic cache characteristics. When simulation begins, variables within cache_t2 are set based on the contents of the corresponding cache_t variables. Thus, the distinction between simulation phase and the rest of the tool provides simplicity.

Moreover, if only cache_t2 was used instead of cache_t for the whole program, then unnecessary memory would be occupied. This is the second reason for the coexistence of both structures. Given that Valgrind developers create tools to increase memory-friendliness, they seriously consider memory wasted, thus, any unnecessary memory occupation is avoided.

3.2.3 cg_pre_clo_init

cg_pre_clo_init is the first function called when Cachegrind is executed and is used for initialisation of details, needs and event tracking. Details allow the tool developer specify the name of the tool, the version number, the name of the author, the license type and other similar kind of information. Needs specify to the core any particular needs of the tool, such as reports of errors detected by Valgrind's core, definitions of its own command line options, etc. Finally, the tool specifies what kind of events it will get notified about from the core. There are some default events every tool gets notified for, but in case they are not enough, they have to be specified manually.

Cachegrind only initialises details and needs, and uses the default event tracking provided by the Valgrind core. Details include the tool name, the version and a description of the tool, while needs specify that Cachegrind defines its own command line options and that information is kept by Cachegrind about specific instructions or translations, which may have to be discarded when translations are unloaded to avoid stale information being reused for new translations.

3.2.4 cg_post_clo_init

cg_post_clo_init is used only if tools provide command line options (clo) and does some further initialisations based on these options. For Cachegrind, cg_post_clo_init's function is crucial. First of all, it creates using `VG_(OSetGen_Create)` the cost centre table (CC_table), the instruction information table (instrInfoTable) and the string table (stringTable). Secondly, it calls a function which is responsible for determining the characteristics (size, associativity and line size) of each cache level using CPUID instructions and stores them in 3 variables of type cache_t representing I1, D1 and LL caches. However, in case that cache characteristics were defined as a command line option, they are used instead of the auto-detected ones. Finally, cg_post_clo_init initialises each cache for simulation.

3.2.5 cg_instrument

The understanding of cg_instrument is closely related to the understanding of the process of instrumentation. Recall that "instrumentation is a technique for inserting extra code into an application to observe its behaviour" [16]. Moreover, recall that Valgrind uses D&R, "where the machine code is converted to an intermediate representation (IR) in which each instruction becomes one or more IR operations. This IR is instrumented (by adding more IR) and then converted back to machine code" [22]. Thus, there is a series of phases performed so that D&R is completed, with instrumentation being one of them. In fact, there are 8 phases, 7 of which are performed by the Valgrind core and only one, instrumentation, is performed by the tool [22]. The tool is passed from the core a VEX IR (which is Valgrind's IR) and produces an instrumented VEX IR, which is later passed back to the core [22]. "The easiest way to instrument VEX IR is to enter calls to C functions when interesting things happen" [20]. This is exactly what cg_instrument does. It looks for memory events and, when they occur, it calls functions to act upon them.

An ordered list is maintained consisting of memory events, for which no IR has yet been generated to do the relevant helper calls. The superblock (recall that IR blocks are called *superblocks* and each superblock represents from 1 to 50 instructions) is scanned top to bottom and memory events are added to the end of the list [20]. "At various points the list will need to be *flushed*, that is, the IR generated from it. Flushing the list consists of walking it start to end and emitting instrumentation IR for each event, in the order in which they appear" [20]. Note that superblocks (type IRSB), among others, contain a list of statements (type IRStmt) which represent code. Statements (IRStmt) represent operations with side-effects, e.g. stores, assignment to temporaries, etc. cg_instrument, depending on the type of each statement, calls the corresponding event function; these are: Instruction Reads (Ir), Data Reads (Dr), Data Writes (Dw), Conditional Branch Counts (Bc) and Indirect Branch Counts (Bi) [20]. When events are flushed, as the list is walked from start to end based on the type of the event, the corresponding helper function calls occur which update data and allow cache simulation to take place.

3.2.6 cg_fini

Recall that cg_fini "is used to present the final results, as a summary of the information collected and write out any log files" [11]. For Cachegrind, statistics for Instruction and Data references are presented, as well as the misses and the miss rate for I1, D1 and LL caches. Moreover, data calculated are written in a file name "*cachegrind.out.<pid>*" (unless specified otherwise by the user), "where *<pid>* is the program's process id" [10]. Data stored in that file can be used for later analysis and annotation.

3.3 L2 Cache Inclusion

This section describes the procedure underneath L2 cache inclusion. It only provides a high-level explanation of the inclusion, since a low-level one that contains all the changes in code would break up the flow of the text and would lose the reader. As a result, the gist behind the inclusion would be difficult to be understood.

As it has been stated in "*3.1.1 — General approach to L2 cache extension*" L2 cache inclusion will be treated in a very similar way that Cachegrind treats LL cache. This has been decided on the basis that the LL cache that Cachegrind already simulates can either represent L2 or L3. Thus, the mechanism to simulate and provide information for the L2 cache is already implemented within Cachegrind and does not have to be created from scratch.

To be more specific the following aspects had to be considered and implemented:

- **Create variables to hold L2 cache information.**
  Variables of type cache_t had to be created to hold L2 cache information (size, associativity and line size). Moreover, when initialisation of the several aspects concerning the cache took place, values were set to be similar to LL cache, but not always identical.

Interestingly, no cache_t2 variables had to be created since Cachegrind initialises the cache simulation using a macro which automatically sets cache_t2 variables to be of the same name as the one used to call the simulation initialisation macro.

- **Extend current structures responsible for holding L1 and LL information, to hold information for L2 as well.**
  In particular CacheCC was extended to include L2 misses.

- **Extend Cachegrind's command line options to allow setting L2 cache manually.**
  A cache_t variable was created to hold the command line specified cache characteristics. Moreover, the appropriate function calls were made to check whether L2 was defined as a command line option, and if yes, ensure the validity of its contents.

- **Initialise L2 cache for simulation.**
  Only required the use of a macro to initialise L2 cache at the same part of the code that initialisation of I1, D1 and LL caches took place. This macro provided great automatisation, so no further initialisations had to be done for simulation.

- **Extend functions to include L2 cache in their parameters.**
  All functions where L2 cache was needed, either to get or to pass information contained within it, were changed to include it. This included, but was not limited to, functions related to CPUID instructions where caches had to be passed as parameters.

- **Change the simulation code to simulate L2 cache as well.**
  The simulation function worked as follows: misses in L1 resulted in the "pass" of data to LL[2]. This was changed so that misses in L1 would result in a pass to L2, which in case of a miss would result in a pass to LL.

- **Present L2 cache results to the user before program terminates.**
  cg_fini was extended to include L2 results.

- **Write L2 cache results on the log file generated by Cachegrind (cachegrind.out.X).**
  The function that writes the log file was extended to write information for L2 cache as well.

3.4 Original's Cachegrind Confusion

As the L2 cache inclusion took place, a confusion was noticed in Cachegrind's initial version (version 3.8.1). For some, Cachegrind does not clarify whether it treats the caches as **a)** first level and last level (i.e. L1 and L2 or L3) or **b)** first level and the later level (i.e. L1 and L2+L3, where "+" refers to L2 and L3 results added together). To be more specific, original Cachegrind produces a result regarding the LL cache, although, it is not clarified whether that result should represent L2 or L3 cache specific-statistics or it represents both of them.

According to the simulation algorithm LL represents the later level, i.e. L2+L3, while explanation given in the specification makes some believe that LL represents either L2 or L3.

3.5 Running Extended Cachegrind

In this section the extended version of Cachegrind is executed, aiming at demonstrating that the version works and at showing its results. However, the validity of the results is neither discussed nor checked here, as this is the subject of a following section. Yet, some logical

[2]There is a confusion regarding the simulation function as it will be seen in the following section "*Original's Cachegrind Confusion*".

assumptions upon the expected results and the results obtained are made here, although they cannot be considered a check upon the validity.

Assuming programs running in 1 core, the expected results (number of cache references and cache misses) of L2 cache should be smaller or equal than L1 and bigger or equal than LL i.e. $L1_{Results} <= L2_{Results} <= LL_{Results}$. This is because if there is a L1 cache miss, then that miss will be looked up in L2 cache which is bigger in size and contains more data, thus chances are increased that a look-up will result in a hit. In the same concept, a L2 cache miss will be looked up in L3 cache. Moreover, the number of references of a particular level of cache should be equal to the number of misses in the previous cache level (this is only true for L2 and L3 caches) i.e. L2 references should be equal to I1+D1 misses, and LL(=L3) references should be equal to L2 misses.

Plenty of programs were executed to ensure that the extended version of Cachegrind works with a variety of programs without errors. However, only two of these executions are shown in this section, since they are of particular interest for reasons that will be explained next. Programs named cg.c and cgslow.c[3] can be seen here, with a comment highlighting their difference, while analysis follows:

Listing 3.9: cg.c

```
1  #define SIZE (2048)
2
3  int main(void){
4
5  int h, i, j;
6  static int a[SIZE][SIZE];
7
8  for (h = 0; h < 10; h++)
9          for (i = 0; i < SIZE; i++)
10                 for (j = 0; j < SIZE; j++)
11                         a[i][j] = 0;
12
13 return 0;
14 }
```

Listing 3.10: cgslow.c

```
1  #define SIZE (2048)
2
3  int main(void){
4
5  int h, i, j;
6  static int a[SIZE][SIZE];
7
8  for (h = 0; h < 10; h++)
9          for (i = 0; i < SIZE; i++)
10                 for (j = 0; j < SIZE; j++)
11                         a[j][i] = 0;    //!!difference. j comes first,↩
                                   then i.
12
13 return 0;
14 }
```

Both programs initialise an array whose memory size is equal to 2048 * 2048 * 4 = 16MB. 2048 are the height and width of the array, while, 4 is the size of each integer assuming 4 byte sized integers. The 16MB array does not fit in the 6MB L3 cache of the computer that executions take place.

[3]Both cg.c and cgslow.c were written by Nicholas Nethercote in [21] and they were slightly changed to fit our needs.

As stated above, the two programs are of particular interest. This is because they do the same operations in a very different way. They both initialise a 2048 by 2048 array to 0, 10 times. However, cg.c performs a row-major traversal [21] and when compiled and executed takes 0.080s ("user time" as measured by `/usr/bin/time` command), while, cgslow.c performs a column-major traversal and takes 0.628s to execute, i.e. 7.85 times slower[4] [21]. In both programs, the 16MB array is too big to fit into any cache level. In cg.c, the traversal is done along cache lines, so only every 16th (64B line size holds 16 4-byte integers) array accesses cause a (D1, L2 and LL) cache miss [21]. In cgslow.c, the traversal is done across cache lines, so every array access causes a cache miss. A 16-fold increase in LL cache misses causes a 7.6-fold increase in execution time [21]. Thus, it is expected that cg.c execution will produce less misses than cgslow.c and will take less time to complete.

3.5.1 Run with cg.c

By compiling cg.c and running it with Cachegrind, we get the following (warnings and results are analysed right after):

Listing 3.11: Cachegrind Execution of cg.c

```
1  $gcc -o cg cg.c
2  $valgrind --tool=cachegrind ./cg
3  ==14388== Cachegrind, a TLB, cache and branch-prediction profiler
4  ==14388== Copyright (C) 2002-2012, and GNU GPL'd, by Nicholas Nethercote et al.
5  ==14388== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
6  ==14388== Command: ./cg
7  ==14388==
8  --14388-- warning: L4 cache ignored
9  --14388-- warning: L3 cache found, using its data for the LL simulation.
10 ==14388==
11 ==14388== I   refs:      293,853,591
12 ==14388== I1  misses:            638
13 ==14388== L2i misses:            634
14 ==14388== LLi misses:            634
15 ==14388== I1  miss rate:        0.00%
16 ==14388== L2i miss rate:        0.00%
17 ==14388== LLi miss rate:        0.00%
18 ==14388==
19 ==14388== D   refs:      209,847,700  (167,869,531 rd + 41,978,169 wr)
20 ==14388== D1  misses:      2,622,582  (        970 rd +  2,621,612 wr)
21 ==14388== L2d misses:      2,622,509  (        905 rd +  2,621,604 wr)
22 ==14388== LLd misses:      2,622,509  (        905 rd +  2,621,604 wr)
23 ==14388== D1  miss rate:         1.2% (        0.0%   +  6.2%  )
24 ==14388== L2d miss rate:         1.2% (        0.0%   +  6.2%  )
25 ==14388== LLd miss rate:         1.2% (        0.0%   +  6.2%  )
26 ==14388==
27 ==14388== L2 refs:         2,623,220  (  1,608 rd +  2,621,612 wr)
28 ==14388== L2 misses:       2,623,143  (  1,539 rd +  2,621,604 wr)
29 ==14388== L2 miss rate:          0.5% (  0.0%     +  6.2%  )
30 ==14388==
31 ==14388== LL refs:         2,623,143  (  1,539 rd +  2,621,604 wr)
32 ==14388== LL misses:       2,623,143  (  1,539 rd +  2,621,604 wr)
33 ==14388== LL miss rate:          0.5% (  0.0%     +  6.2%  )
```

3.5.2 Warning Message Analysis

All the same warning messages are present in the initial version of Cachegrind. They are not side-effects of the extensions made. An analysis of each of the 3 warning messages follows

[4] Nearly the same test was initially done by Nicholas Nethercote in [21], but for a 4MB array, in 2 levels of cache, where he found a decrease of 12.5 times.

in order of appearance.

1. **Unknown Intel cache config value (0x63), ignoring**
   Recall that file cg-x86-amd64.c is responsible for detecting, using the CPUID instruction, the characteristics of the underlined hardware. At the moment, this is used to identify characteristics of CPU caches and does not include the TLB. 0x63 is hex code for a particular TLB configuration, especially corresponds to data TLB with 1G pages, 4-way associativity and 4 entries. Cachegrind cannot detect it, since it does not include TLB thus a warning message is raised.

   A question should be raised at this point regarding why does only one TLB configuration raise a warning and not more. Intel CPUID instruction requires indexing, i.e. CPUID returns a configuration code which represents a particular cache or TLB configuration (cache size, cache associativity and cache line size or TLB page size, TLB associativity and TLB total entries) and in order to find the exact configuration a switch statement is used which contains all or a subset of the configurations. If the given configuration matches a switch case statement, some appropriate action is taken, otherwise the *default* case raises the message "*Unknown Intel cache config value (X), ignoring*", where X is the given configuration value. In order for Cachegrind not to produce warnings for every TLB configuration, switch cases cover many of the TLB configurations, without taking any action. They are only to there so the default case is not reached. Otherwise, many warnings would be produced. By the time Cachegrind version 3.8.1 was written more configurations have been added by Intel. Thus, no switch case statement exists for them and the default value is used. This is what happened in this warning case. The following code snippet shows the existing TLB switch cases and the default case as explained above. Moreover, two I1 cache configurations are included (cases 0x06 and 0x08) to show what happens in a relevant CPUID configuration code.

```
1  switch (...) {
2
3        ....(code not shown)
4
5        /* TLB info, ignore */
6        case 0x01: case 0x02: case 0x03: case 0x04: case 0x05:
7        case 0x0b:
8        case 0x4f: case 0x50: case 0x51: case 0x52: case 0x55:
9        case 0x56: case 0x57: case 0x59:
10       case 0x5a: case 0x5b: case 0x5c: case 0x5d:
11       case 0x76:
12       case 0xb0: case 0xb1: case 0xb2:
13       case 0xb3: case 0xb4: case 0xba: case 0xc0:
14       case 0xca:
15           break;
16
17       case 0x06: *I1c = (cache_t) {  8, 4, 32 }; break;
18       case 0x08: *I1c = (cache_t) { 16, 4, 32 }; break;
19
20       ...(more cache configurations not shown)
21
22       default:
23          VG_(dmsg)("warning: Unknown Intel cache config value (0x↩
               %x), ignoring\n",
24                    info[i]);
25          break;
26 }
```

2. **L4 cache ignored**
   The computer where the above execution took place has 4 levels of CPU caches, however, Cachegrind can only handle 3 levels of caches. Thus, a warning message was raised to inform the user.

3. **L3 cache found, using its data for the LL simulation.**
   This warning is present in all Cachegrind executions on computers that have 3 levels of CPU caches. It informs the user that 3 levels of CPU caches were found and that LL would represent L3.

3.5.3 Result Analysis

First of all, by examining the number of the LL references (2,623,143) and the LL misses (2,623,143) we conclude that the array did not fit in LL(=L3) cache and resulted in 100% misses. As we can see, results for L2 cache satisfies our range expectation i.e. $L1_{Results} <= L2_{Results} <= LL_{Results}$, where *Results* refer to misses. Particularly:

- I1 misses=638 <= L2i misses=634 <= LLi misses=634
- D1 misses=2,622,582 <= L2d misses=2,622,509 <= LLd misses=2,622,509

Moreover, for L2 and L3 caches, the number of references in either level is equal to the number of misses in the previous level:

- I1misses+D1misses=2,623,220 (638+2,622,582) which is equal to L2 references=2,623,220.
- L2 misses=2,623,143 which is equal to LL references=2,623,143

Furthermore, the limit value for value h has a crucial effect on the number of data misses produced in all caches, however, in all cases the percentage remained 100%. h specifies how many times all the array values will be set to 0 (recall cg.c or cgslow.c code). As h changes the number of data misses change almost h times.
If h=0, then

- D1 misses=263,277
- L2d=263,204
- LLd misses=263,204

If `for(h=0;h<5;h++)`, then

- D1 misses=1,311,857 (= 4.98 * h(0)[5])
- L2d misses=1,311,784 (= 4.98 * h(0))
- LLd misses=1,311,784 (= 4.98 * h(0))

Finally, if `for(h=0;h<10;h++)`, then

- D1 misses=2,622,582 (=9.96 * h(0))
- L2d misses=2,622,509 (=9.96 * h(0))
- LLd misses=2,622,509 (=9.96 * h(0))

The above observation is as expected. Assuming that the total number of data processed is h, every time the loop iterates, the total number of data increases by h.

[5] h(0) refers to h=0 corresponding cache misses

3.5.4 Run with cgslow.c

By compiling cgslow.c and running it with Cachegrind, we get the following (warnings and results are analysed right after):

Listing 3.12: Cachegrind Execution of cgslow.c

```
1  $gcc -o cgs cgslow.c
2  $valgrind --tool=cachegrind ./cgs
3  ==15065== Cachegrind, a TLB, cache and branch-prediction profiler
4  ==15065== Copyright (C) 2002-2012, and GNU GPL'd, by Nicholas Nethercote et al.
5  ==15065== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
6  ==15065== Command: ./cgs
7  ==15065==
8  --15065-- warning: L4 cache ignored
9  --15065-- warning: L3 cache found, using its data for the LL simulation.
10 ==15065==
11 ==15065== I   refs:      293,853,603
12 ==15065== I1  misses:            638
13 ==15065== L2i misses:            634
14 ==15065== LLi misses:            634
15 ==15065== I1  miss rate:        0.00%
16 ==15065== L2i miss rate:        0.00%
17 ==15065== LLi miss rate:        0.00%
18 ==15065==
19 ==15065== D   refs:      209,847,700  (167,869,532 rd + 41,978,168 wr)
20 ==15065== D1  misses:     41,944,172  (        960 rd + 41,943,212 wr)
21 ==15065== L2d misses:     41,944,100  (        895 rd + 41,943,205 wr)
22 ==15065== LLd misses:     41,944,100  (        895 rd + 41,943,205 wr)
23 ==15065== D1  miss rate:        19.9% (        0.0%   + 99.9%  )
24 ==15065== L2d miss rate:        19.9% (        0.0%   + 99.9%  )
25 ==15065== LLd miss rate:        19.9% (        0.0%   + 99.9%  )
26 ==15065==
27 ==15065== L2 refs:        41,944,810  (  1,598 rd + 41,943,212 wr)
28 ==15065== L2 misses:      41,944,734  (  1,529 rd + 41,943,205 wr)
29 ==15065== L2 miss rate:          8.3% (  0.0%     + 99.9%  )
30 ==15065==
31 ==15065== LL refs:        41,944,734  ( 1,529 rd + 41,943,205 wr)
32 ==15065== LL misses:      41,944,734  ( 1,529 rd + 41,943,205 wr)
33 ==15065== LL miss rate:          8.3% ( 0.0%     + 99.9%  )
```

3.5.5 Warning Message Analysis

Warning messages and their explanation is the same as in previous execution.

3.5.6 Result Analysis

Again, by observing the number of LL refs and LL misses we conclude that the array did not fit in L3 cache. As we can see, results for L2 cache satisfies our range expectation, i.e. $L1_{Results} <= L2_{Results} <= LL_{Results}$ where *Results* refer to misses. Particularly:

- I1 misses=638 <= L2i misses=634 <= LLi misses=634
- D1 misses=41,944,172 <= L2d misses=41,944,100 <= LLd misses=41,944,100

Moreover, for L2 and L3 caches, the number of references in either level is equal to the number of misses in the previous level:

- I1misses+D1misses=41,944,810 (638+41,944,172) which is equal to L2 references=41,944,810.
- L2 misses=41,944,734 which is equal to LL references=41,944,734

Furthermore, the h effect is present again. This time the change in data misses that the limit of h caused, is closer to h than previously. Previous values were 4.98 and 9.96, while now 4.99 and 9.99 were noticed.

3.5.7 Conclusion on Extended Version Execution

The extended version of Cachegrind was executed and worked, without producing any side-effects. Moreover, results were in line with the expected outcome. However, this does not draw any conclusions upon their validity, which is discussed in the next section.

3.6 Validity of Results

This section deals with whether the statistics gathered by the extended version of Cachegrind for L2 cache are correct or not. After describing the ways devised to draw upon validity of statistics, it focuses on their implementation. In addition, the section ensures that no changes occurred in L1 results. L3 results cannot be verified since, as described in section "*3.4 — Original's Cachegrind Confusion*" they are confusing.

3.6.1 Validation Methods

Three ways were devised in order to draw upon validity of statistics. However, only the first two described here were implemented. The first way requires altering the code of the initial Cachegrind version so that, in case 3 level of caches are encountered, it will not keep L3 characteristics and drop L2 characteristics but, instead, it will drop L3 and keep L2. Thus, the results printed afterwards for LL cache will actually represent L2 cache even if 3 levels of cache are found. A simple comparison between those results and the extended's version is enough to know whether the extension is valid or not. If results are the same, then everything is fine, otherwise we should consider what went wrong.

The second way is to set the characteristics of LL cache to be the same as those of L2 cache. Cachegrind allows to set manually the characteristics of all caches instead of auto-detecting them. If LL is set to have the characteristics of L2 cache, then Cachegrind would actually simulate L1 and L2 caches. Again, a simple comparison between those results and those of the extended version is enough to know whether the extension is valid or not. If results are the same, then everything is fine, otherwise we should consider what went wrong. In effect, the first and the second way are the same. However, the first way requires code alteration, while the second does not.

The third way is to reason upon the validity of the statistics is to prove mathematically that results are valid. However, it was decided not to proceed with it, since it depends upon on a number of factors that are very difficult to be conceptualised and estimated. Thus, the first two ways are only consider to verify the results of the extension.

The next sections explain the implementation of the two ways used to check and draw upon the validity of the results.

3.6.2 Code Alteration LL=L2

Within this section, the first way to draw upon the validity of results is considered and implemented. Cachegrind, in cg-x86-amd64.c, which is responsible for detecting cache characteristics, uses the boolean *L3_found* to check if any CPUID instruction matches a L3 cache configuration. If it does, then L3_found is set to true. Moreover, a local variable *L3c* is created to hold the L3 cache characteristics and LLc, which is passed as a parameter, holds the L2 cache characteristics. After all CPUID configurations have been processed, Cachegrind, checks if L3_found is true. If yes, then it sets LLc to contain L3 characteristics and informs the user about it. Contents of LLc are those that will be used in simulation. By changing the code so that LLc keeps the characteristics of L2, we make Cachegrind simulate the L2 cache.

This is done by putting in comments the part of the code that sets the LL cache to be equal to the characteristics gathered for L3 cache. This part of code exists in cg-x86-amd64.c and is the following:

Listing 3.13: -

```
1   /* If we found a L3 cache, throw away the L2 data and use the L3's ↩
       instead. */
2    if (L3_found) {
3       VG_(dmsg)(  warning  : L3 cache found, using its data for the LL↩
            simulation.\ n  );
4       *LLc = L3c;
5       L2_found = True;
6    }
```

Which was altered to this one:

Listing 3.14: -

```
1   /* If we found a L3 cache, throw away the L2 data and use the L3's ↩
       instead. */
2    if (L3_found) {
3       //VG_(dmsg)(  warning  : L3 cache found, using its data for the ↩
           LL simulation.\ n  );
4       //*LLc = L3c;
5       L2_found = True;
6    }
```

Now, we proceed to compiling and running the initial version of Cachegrind with only change the one mentioned above. Many programs were tested to ensure that results are valid. However, only results from programs cg.c and cgslow.c are shown in this section.

Outputs are not shown here since they would break up the flow of the text. However, data have been gathered in the following table. Columns "*Init. (LL=L2)*" stand for "Initial version" and refer to results obtained from Cachegrind executions where LL cache represented L2 cache. "*Extended*" columns stand for "Extended version" and refer to results obtained from the extended version of Cachegrind.

	Init. (LL=L2)	**Init. (LL=L2)**	**Extended**	**Extended**
	cg	**cgs**	**cg**	**cgs**
I refs:	73,581,182	73,581,198	73,581,184	73,581,198
I1 misses	729	731	729	731
L2 misses	-	-	**632**$_1$	**634**$_2$
LLi misses	**632**$_1$	**634**$_2$	627	629
I1 miss rate	0%	0%	0%	0%
L2i miss rate	-	-	**0%**$_3$	**0%**$_4$
LLi miss rate	**0%**$_3$	**0%**$_4$	0%	0%
D refs	52,520,411	52,520,411	52,520,411	52,520,411
D1 misses:	656,482	10,486,762	656,482	10,486,762
L2 misses	-	-	**656,414**$_5$	**10,486,695**$_6$
LLd misses:	**656,414**$_5$	**10,486,695**$_6$	66,481	66,481
D1 miss rate:	1.2%	19.9%	1.2%	19.9%
L2 miss rate	-	-	**1.2%**$_7$	**19.9%**$_8$
LLd miss rate:	**1.2%**$_7$	**19.9%**$_8$	0.1%	0.1%
L2 refs:	-	-	**657,211**$_9$	**10,487,493**$_{12}$
L2 misses:	-	-	**657,046**$_{10}$	**10,487,329**$_{13}$
L2 miss rate:	-	-	**0.5%**$_{11}$	**8.3%**$_{14}$
LL refs:	**657,211**$_9$	**10,487,493**$_{12}$	657,046	10,487,329
LL misses:	**657,046**$_{10}$	**10,487,329**$_{13}$	67,108	67,110
LL miss rate:	**0.5%**$_{11}$	**8.3%**$_{14}$	0%	0%

As we can see from the above results, data that represent LL (i.e. L2) in the initial version columns are the same with those representing L2 in the extended version columns (values in bold, subscripts also help to understand which values should -and indeed- match). Thus, it has been checked that the extension produces correct L2 results.

3.6.3 Manually Set LL=L2

The notion behind this way of checking the validity of results is to set in the initial version of Cachegrind (version 3.8.1, without any code alterations) the LL cache characteristics to be the same as those of L2. Thus, Cachegrind would actually simulate L1 and L2 instead of L1 and L3. The computer in hand has a L2 cache of 256KB (262144 B) size, 8-way associative with 64KB line size.
In order to run the initial version of Cachegrind with the above settings, the following command must be executed:
`valgrind --tool=cachegrind --LL=262144,8,64 ./X`
where X represents a given compiled program.

Again, many programs were tested to ensure that results are valid. Results from programs cg.c and cgslow.c are exactly the same with those presented in the table of the previous subsection (last table used) under the "Init. (LL=L2)" column. Thus, we checked again, using a second way, that the extension produces correct L2 results.

3.7 Measuring Execution Time

One of the initial requirements set before extending Cachegrind was not to introduce a major slowdown in its performance. Now that L2 cache inclusion has been completed, some tests should be performed to ensure that the extensions are efficient and execution time is close to the initial version. Execution Time measurement was done by timing execution of 10 programs 10 times and calculating their average. Tests were performed using both Cachegrind's initial and extended version and execution time was measured using the built-in Unix program `/usr/bin/time` ("user time"). Results have been collected and are shown in Figure 3.3. The X axis represents different program executions, while, the Y axis shows their corresponding (averaged) timings. As it can be seen, the extended timings (shown in red) are higher than the initial timings (shown in blue) i.e. the extended version took more time to execute. Most of the times the extended version execution time was very close to the initial, while others, it deviated (e.g. executions 2 and 5).

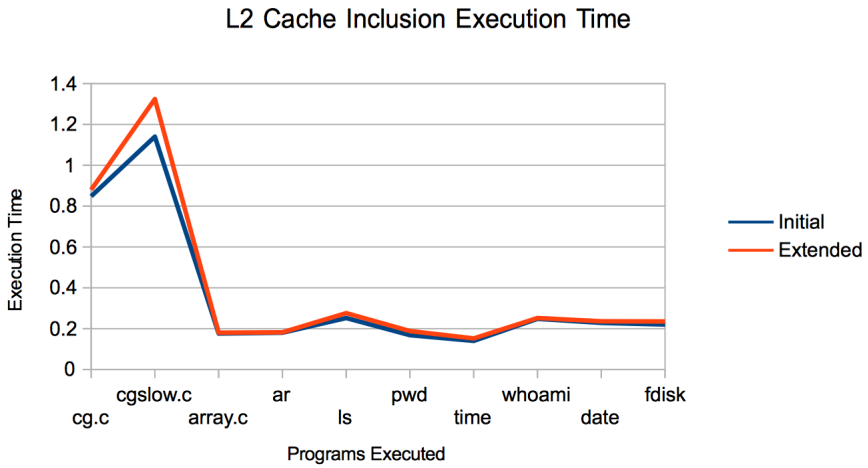

Figure 3.3: Initial & Extended Execution Times

All executions, apart from cg.c, cgslow.c and, array.c, refer to Unix commands. Moreover, array.c was not mentioned before and its source code be seen here:

```
1  #define SIZE (100)
2  int main(){
3
4              int i,array[SIZE];
5
6              for(i=0;i<SIZE;i++){
7                 array[i]=i+10;
8              }
9
10           return 0;
11 }
```

Execution 2 (cgslow.c) is the one that deviates the most. This is probably due to the fact that the executed program was not cache friendly. Recall from section "*3.5 — Running Extended Cachegrind*" that cgslow.c produces many misses and takes a large amount of time to run. By adding extra data measurement (L2 cache data measurement), execution time is further increased.

Excluding execution 2 whose slowdown was 16%, all other executions (i.e. of "normal" programs) did not introduce a slowdown greater than 10%.

3.7.1 Conclusion on Time

As results indicated, the extended version of Cachegrind takes more time to execute. In general, execution time is close to the initial version and does not introduce a slowdown greater than 10% of the initial execution time, except in cases where the program in hand is cache unfriendly, where a slowdown of 16% was noticed.

Chapter 4

TLB Measuring

4.1 Introduction

This chapter focuses on TLB measuring and describes the way that Cachegrind was approached and extended to provide information for TLB. It starts by explaining the concepts within TLB is understood, such as virtual memory and paging. The chapter then discusses basic TLB underlined notions, such as address translation, a simple TLB algorithm, as well as an example to illustrate TLB's operation. After that, the capabilities and the characteristics of the extension are described and an in depth explanation is provided regarding the extension's design and implementation. Moreover, the extension is executed, results are shown, and the program options are explored in order for the reader to familiarise himself with the extension. As a final step, the validity of the results is checked and a comparison of the execution times between the original version and the extended one is performed aiming to observe the effect of the extensions on execution speed.

As before, all of the sections focus on providing high-level explanations, however, sometimes, low level explanations, i.e. C code, are provided to ease understanding, but they are limited. Finally, this project is only concerned to make any extensions work in an Intel x86 machine. As a result, all of the extensions described in this chapter are Intel x86 specific.

4.1.1 Virtual Memory, Paging and the TLB

As it has already been stated, TLB is a cache of popular virtual to physical addresses. Virtual addresses are generated due to the concept of virtual memory that lies underneath modern OSs. Due to reasons regarding security and efficiency, the OSs provide a "virtual world" — called *address space* — for programs to execute, where each program has its own view of memory and it cannot read or write other programs' address spaces. This is possible due to the OS kernel that creates what is known as *virtual memory*, which is nothing more than an abstraction of the physical memory that every program running in *user-mode* can see. Since virtual addresses (virtual memory addresses) are merely an abstraction, they cannot be used to access real physical memory locations. Thus, when it is time for a program to execute, the kernel gets every virtual address that the program generates and translates it to a corresponding physical address which can be used to access memory. Our view of virtual to physical translation will be extended after explaining *paging*.

Paging is a technique to virtualise memory, by splitting the address space into fixed-sized units called *pages* [23]. The physical memory is also split into some number of same sized pages [3]. To distinguish between address space pages and physical memory pages, the terms *virtual page* and *page frame* are used respectively.

In order for a translation to occur, the OS has to know where each virtual page should be placed in physical memory. As a result, a list of all the virtual pages with their corresponding page frames is held. That list is known as the *page table*. Note that page tables are created per process, since each process has its own virtual to physical correspondences. The main disadvantage of the page tables is that they are saved in main memory and, as it has already

been indicated in the "*Literature Review*" chapter of this study, accessing main memory is slow and several ways have been devised to avoid it, mainly in the form of caches. The TLB is a cache that holds a small portion of the page table with the most popular translations. A simplified view of the TLB can be thought of a table with two columns and a number of rows representing the total number of entries. One column holds the virtual pages, while the other, the physical[1]ones. A simplified view of TLB is illustrated in Figure 4.1. When time for a translation comes, the virtual address generated by the program is looked up in the column of the TLB that holds the virtual addresses. If a match occurs, then the corresponding physical address is used for the translation. Otherwise, the page table located in main memory has to be looked up for the translation to occur.

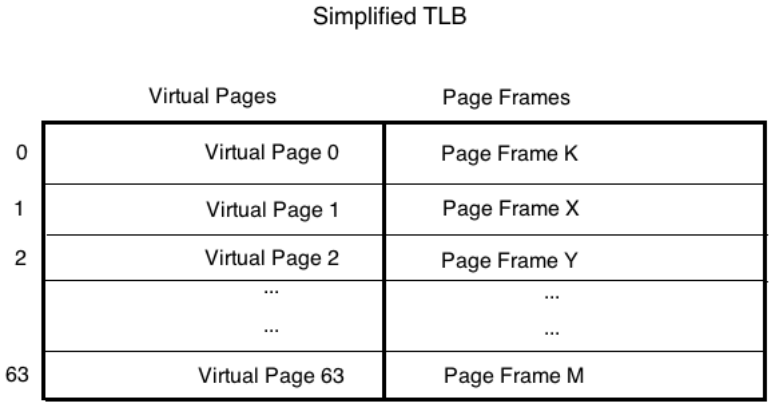

Figure 4.1: A 64-entry simplified TLB

4.1.2 Address Translation

Given a virtual address, to translate it to its corresponding physical address, we need to split the address into two components: the *Virtual Page Number* (VPN) and the *page offset* (offset). The common scheme used is that the upper bits of an address represent the VPN, while the lower bits of the address represent the offset. This scheme is also used by the x86 Intel processors [6] and thus, it will be used. The offset part is the same in both virtual and physical addresses, thus, the only aspect needed for the translation to occur, is the translation of the VPN into physical page number (PFN). The address space size and the page size have a crucial role on how is a virtual address split into VPN and offset. Given a virtual address space size K in bits and a page size PG in bytes, we can calculate the VPN and the offset bits as follows:

$$offset\_bits = log2(PG)$$
$$VPN\_bits = K - offset\_bits$$

Figure 4.2 illustrates the concept of VPN and offset split in a given virtual memory address.

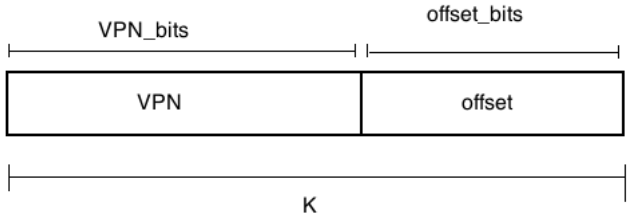

Figure 4.2: A virtual address of K size is split into VPN and offset

[1]The terms "*page frame*" and "*physical frame*" are used interchangeably.

An example follows to demonstrate the above. Assume a 32 bit space with 4KB page size. Then $K = 32$ and $PG = 4 * 1024 = 4096$ bytes. Thus:

$$offset\_bits = log2(4096) = 12\ bits$$

$$VPN\_bits = 32 - 12 = 20\ bits$$

Then the number contained from bit $K-1$ to bit $K - VPN\_bits$ is looked up in the TLB and the page tables for its corresponding PFN. Figure 4.3 illustrates the process of translation assuming a smaller physical address space than the virtual address space.

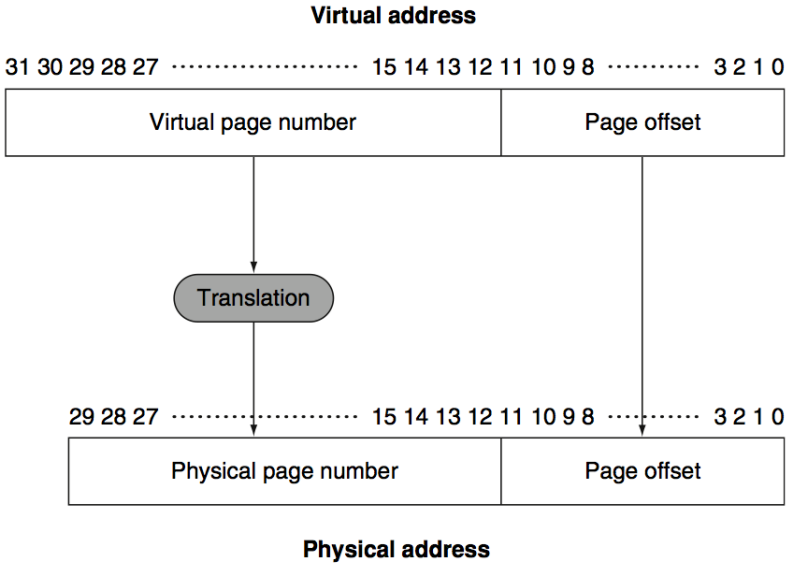

Figure 4.3: Address Translation [23]

4.1.3 Basic TLB Algorithm

This section describes a basic and simplified algorithm used for address translation. Given a generated virtual address, the VPN should be extracted and looked up in the TLB. In case of a TLB hit, the PFN that matches the VPN is obtained and the translation can be completed. Otherwise, in case of a miss, the VPN should be looked up in the page table. A pseudocode version of this algorithm has as follows:

```
1  vpn= (virtual address & vpn_mask ) >> shift_by_offset_bits;
2
3  if( tlb_lookup(vpn) == true){
4          hit++;
5          do_translation();
6  }
7  else{
8          miss++;
9          page_table_lookup(vpn);
10         load_vpn_to_tlb(vpn);
11 }
```

Since the first line is a bit complicated and was not explained before, an example will be used to illustrate its purpose. Assuming an 8-bit address space, with 8 (=2^3) byte pages and a virtual address with value 117 (01110101_2), by calculating the VPN and offset bits as described before, a 3-bit offset and a 5-bit VPN is obtained. Figure 4.4 illustrates the contents of the virtual address and how is it split into 5-bit VPN and 3-bit offset.

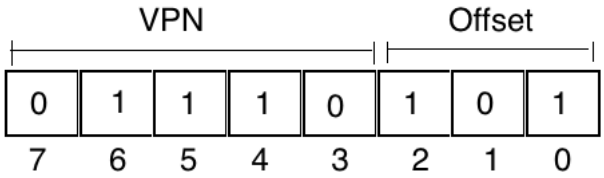

Figure 4.4: Virtual Address 117 (01110101_2)

In order to obtain only the number represented in the 5 VPN bits, the address has to be masked, i.e. ANDed[2] with a number of the same size (8 bits in total), where the 5 most significant bits would be equal to 1 and the rest equal to 0. This is as follows:

```
1      0 1 1 1 0 1 0 1 (117 in base10)
2  AND 1 1 1 1 1 0 0 0 (248 in base10)
3    = 0 1 1 1 0 0 0 0 (112 in base10)
```

The resulting number (112_{10}) is not complete. That number is not VPN yet. It is the virtual address masked which contains the VPN and an offset whose value is 0. The VPN is only the 5 most significant bits of this number. Therefore, in order to extract the VPN, the number is shifted by the offset_bits (=3), resulting in the VPN.
`01110000 >> 3 = 01110` (14_{10})

As it will be explained later, TLB measuring extension was built based on the logic of this basic TLB algorithm.

4.1.4 TLB Operation Example

The operation of the TLB can be seen from an example discussed in [3]. First of all, assume an 8-bit virtual address space with 16 byte pages. Based on the equations described previously, every address should be split into 4-bit ($2^4 = 16$) VPN and 4-bit ($2^4 = 16$) offset. Secondly, assume an array of 10 4-byte integers starting at virtual address 100[3].

Figure 4.5 illustrates how is the array placed inside the address space. The address space has 16 VPNs (the maximum that can be represented with 4-bit VPN) and 16 byte pages. As said, the array starts at virtual address 100, that is, 100/16=6 thus, at VPN=6, with remainder 100%6=4 thus, offset = 4. Only 3 4-byte integers (a[0] ... a[2]) can be placed from virtual address 100 to 112, where the page ends (VPN=6). "The array continues on the next page (VPN=7), where the next four entries (a[3] ... a[6]) are found. Finally, the last three entries of the 10-entry array (a[7] ... a[9]) are located on the next page of the address space (VPN=8)" [3].

[2]Perform a bitwise AND operation.
[3]Decimal numbers are used instead of hexadecimal to ease explanation.

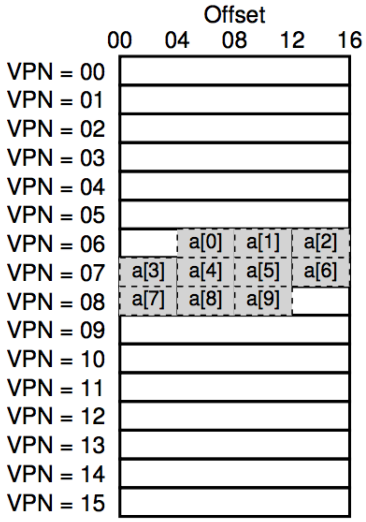

Figure 4.5: Array inside an 8-bit address space, with 16 byte pages [3]

Now that we have seen how the array is placed, assume C code that initialises all the array elements to 0 in the form of a loop as follows:

```
1 int i;
2 for(i=0;i<10;i++)
3         a[i]=0;
```

For simplicity reasons, assume that the only memory accesses generated are to the array, without taking under consideration the generated instructions for the variable "i" and for the "for loop".

The first element accessed is a[0] at virtual address 100. After extracting the VPN and the offset from this address, the values 6 and 4 are obtained respectively. Thus, VPN=6 and offset=4. However, offset is of no importance here, since the focus is to find the corresponding PFN for the given VPN and the offset cannot be helpful at this point. In order to find the PFN, the VPN is passed in the TLB. Given that this is the first time the array is accessed, the TLB will result in a miss, and the page (VPN=6) will be loaded into TLB. Since VPN=6 was just loaded in the TLB, the second array access (a[1]) will result in a hit and so will the the third access (a[2]). However, the fourth access (a[3]) will result in a miss, since the page (VPN=7) has not been loaded into TLB yet, but is loaded immediately after the miss occurs. Thus, a[4], a[5] and a[6] will result in a hit. However, a[7] is in a different page, so a miss will occur again. Finally, array accesses a[8] and a[9] will also result in a hit.

Summarising the TLB activity, 3 misses and 7 hits were noted. Thus, the hit ratio is 70%. Spatial locality had a great impact on TLB's performance. Array elements are "packed tightly into pages (i.e. they are close to one another in space), and thus only the first access to an element on a page yields a TLB miss" [3]. Moreover, if the program accesses the array soon enough that TLB pages have not been overwritten, it is expected that the hit ratio will be higher, probably 100%. This is due to temporal locality, "the quick re-referencing of memory items in time" [3]. Finally, the page size had a significant role on the number of misses. If the page size was bigger, fewer misses would have been expected as more elements of the array would be stored in the same page.

The TLB and address space described above are fictitious, in terms that no modern computer uses such configurations and were only chosen to be such in order to simplify explanation. The norm for the address space is 32 or 64 bits, while the norm of TLBs page size is 4KB. This example was only provided to explain how TLB works.

4.2 TLB Simulator

A TLB simulator[4] was created and integrated into Cachegrind in order to measure the TLB. Initially, the simulator was developed as an autonomous Valgrind tool, because the code in hand would be smaller and no confusion with Cachegrind dependencies would take place during development. When that was completed, it was integrated into Cachegrind. Since all Valgrind tools have a basic structure that should follow, integration was not difficult. This section describes the design of the TLB simulator and the notions that underline it.

The product is not a complete simulator of the TLB. It only simulates those parts of the TLB that are of significant importance and can provide helpful statistics to the developers that use the tool to improve their programs in terms of cache-friendliness. A real TLB simulator would be very difficult to be developed, since it would have to run in kernel-mode and access kernel commands. Our TLB simulator, simulates the TLB with information only available in user-mode. An analysis of its capabilities and characteristics follows.

4.2.1 Capabilities

Cachegrind, after the TLB simulator has been integrated[5] is capable of the following 3 tasks:

1. To measure the number of TLB hits and misses for instruction TLB (iTLB), data TLB (dTLB) and level 2 TLB (L2TLB).

2. To track all the pages used and the times they were accessed per TLB and show them to the user.

3. To provide per function and per source code line statistics regarding TLBs in general, iTLB, dTLB, and L2TLB. However, this task does not occur immediately. As it has already been stated, Cachegrind produces an output file every time it is executed which can be used later for further analysis and annotation. Annotation is done by executing the "cg_annotate" script onto that file and only if this is done, per function and per line of source code statistics are shown to the user. Note that this is along the lines of how the original version of Cachegrind already works and is not a special characteristic of the extension.

4.2.2 Characteristics

Characteristics refer to Cachegrind's available options and to the basic underlined notions of the simulation. Note that many of these will be described later in more depth, however, their gist can be seen here.

1. The user is able to specify what kind of profiling Cachegrind should perform. Among the already existing options, the user can choose to simulate the TLB and/or get information about the pages used and their frequency of use.

2. For Intel x86 computers, the TLB simulator is able to detect, using CPUID instructions, the hardware characteristics of each of the TLBs automatically. The user does not have to enter them manually. Although, that option is also available if needed.

3. Cachegrind allows the user to switch between three cache replacement algorithms. These are *Least Recently Used*, *Least Frequently Used* and *Random*. The first, when it is time to replace contents within a TLB, removes the one used the least recently. The second algorithm, removes the entry used the least frequently, while the third one, removes an entry randomly. Since many real TLBs implement LRU, it was chosen to be the default replacement algorithm.

[4] The terms "*TLB measuring*" and "*TLB simulator*" refer to the same process. A TLB simulator is able to measure the TLB.

[5] From now on, whenever the term "Cachegrind" is used, it refers to the extended version that includes the TLB simulator, unless specified otherwise.

4. Cachegrind assumes a flushed TLB. When the simulation begins, all TLBs being simulated are empty, no entries exist. Knowing the contents of a TLB before simulation would slightly increase the accuracy of the statistics. Besides that, this kind of information is not available in user-mode, so a flushed TLB was used instead.

5. Cachegrind does not account for context switches while simulation takes place. While Cachegrind profiles an application, the kernel may pause it and switch to another process. This action, depending on the hardware vendor and the underlining architecture, may either flush the TLB or with the aid of an address space identifier (ASID) allows context switching to take place causing less harm. In either case, context switches alter the contents of TLB and when execution returns, TLBs' contents are different. Cachegrind assumes that any application runs with no context switches. Note that this is along the lines of how the original Cachegrind works. Context switches may lead to alteration of contents of CPU caches, however, the original version of Cachegrind assumes that any application runs with no context switches.

6. Cachegrind allows the user to set the size of the virtual address space to use in bits. Note that most of the times, when simulating 64 bit address space, the OS actually uses a 48-bit address space. In that case Cachegrind will use the input given by the user. This is why Cachegrind does not provide a check on whether the value set as the size of the virtual address space is a power of 2 or not.

7. The TLB measuring extension of Cachegrind treats which instructions or data to "send" to iTLB or dTLB in exact the same way that the original version of Cachegrind decides how to send instruction or data to instruction and data caches. The reason for that is explained later.

8. Instructions and data are initially looked up in iTLB and dTLB. In case of a miss, they are looked up in the unified L2TLB (if it exists). In case of a L2 TLB miss, instruction or data are cached in both L1 TLBs (iTLB or dTLB) and L2 TLB. Thus, L2 TLB is inclusive.

9. Cachegrind assumes the program is run in a single core and does not account for parallel computing. Moreover, in reality, in Intel x86 computers, L2TLB is shared among the cores, and as a consequence its contents may change by other programs running on a different core. Cachegrind assumes that L2 TLB contents can only change by the running program. This is again along the lines of how the original version of Cachegrind already works. It does not account for parallel computing by default, and it assumes that CPU cache contents can only change by the running program.

10. When the hardware searches TLBs it often does it in parallel. Cachegrind provides a sequential search. Although, this does not affect results and it is merely being mentioned since it is considered to be a characteristic.

11. The page size cannot be auto detected, since it is not accessible in user mode. Thus, the default value set is 4KB i.e. 4096B. However, the user can specify it manually. Moreover, related to this, when large page sizes are used (e.g. 2MB, 4MB or 1GB pages), TLBs for large pages might be used, which are not auto detected and have to be manually set by the user. All TLBs that are auto detected are relevant to 4KB page size.

4.3 Design & Implementation

The notion behind the TLB simulator is very simple. A set of data structures needs to be created to hold information about profiled specific memory events that take place and their results. Cachegrind, with the aid of Valgrind core, already traces all memory accesses which can be categorized as follows: instruction reads (Ir), data reads (Dr), data writes (Dw) and data modifications (read then write) (Dm) [20]. Two questions are raised at this point: a)

Cachegrind already separates events in a specific way in order to "send them" to either I1 or D1. Should the same event separation be used for iTLB and dTLB, and b) to what extent should TLB simulation interfere with the existing Cachegrind code and how should it work?

The answer to the first question can be given by examining the cache arrangement on the latest Intel processors. Both iTLB and dTLB are placed before the level 1 instruction and data caches. This can be seen in Figure 4.6. This configuration is logical, since addresses need to be translated from virtual ones to physical first, in order to be used by the hardware to reference memory addresses. Thus, the TLB simulator can use exactly the same event separation with the rest of Cachegrind. That is, the same instructions that Cachegrind chooses to send in level 1 instruction cache can be sent in iTLB, and the same data that is sent to level 1 data cache can be sent in dTLB.

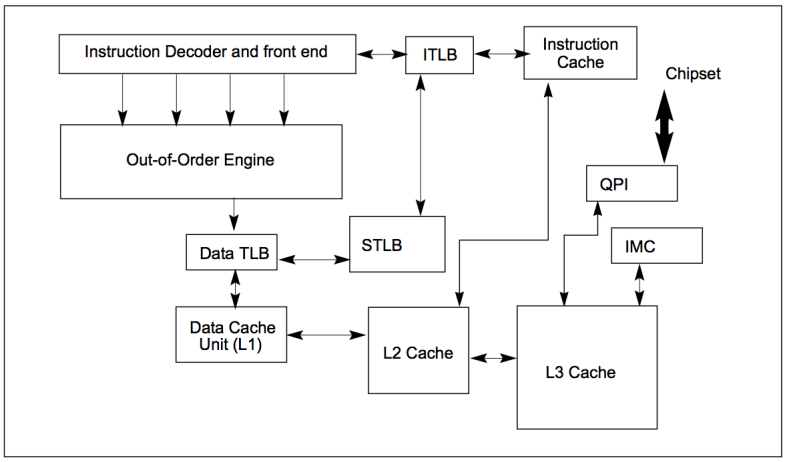

Figure 4.6: "Cache Structure of the Intel Core i7 Processors"[6]

The second question is slightly more complicated and cannot be answered that fast. Regarding the extent to which the TLB simulator should interfere with Cachegrind, it was decided to make it as separate as possible. A new .c file, named cg_tlb.c, containing the TLB simulation would be created and other parts of Cachegrind would have the minimum communication possible with it. This isolated approach was chosen to avoid code and dependencies confusions as well as to keep the extension as simple as possible. Moreover, this is along the lines of how CPU cache is structured: the simulation part is housed in a different file than the main tool. Cachegrind's main file would interact with the TLB simulator only when necessary, through a particular set of functions that notify the simulator for particular events. These include setting up cache configurations, initialising simulation, passing memory events and the corresponding virtual addresses, as well as printing the final results. An explanation of how the simulator works can be seen in the following section.

4.3.1 TLB

TLB measuring is the main aspect of this extension. To measure TLBs three things had to be created:

1. Variables to hold TLB hits and misses per TLB.
2. A structure to represent each of the TLBs.
3. A structure to hold the characteristics of each TLB (iTLB, dTLB and L2TLB).
4. A TLB simulation algorithm.

An analysis of the above follows.

1. **Hit & Miss Variables**

   This is the simplest of the four aspects mentioned. Two variables had to be created per TLB, one for the number of hits and one for the number of misses. Hits and misses were initially created as six global variables, two variables (one for hits and one for misses) for each of the 3 TLBs. However, as development progressed it was decided that it is better to put them in the structure that holds the characteristics of each TLB. This change made the code more compact, more efficient and easier to be expanded, since all information regarding a TLB would be gathered together. Moreover, due to the way that the TLB characteristics structure is built (which will be explained later), it would be easier and would require less lines of code to access hits and misses. Finally, if at a later stage a new TLB had to be added, variables for hits and misses would automatically be created with the '*to hold its characteristics*' structure.

2. **TLB Representation Structure**

   As it has already been stated, a real TLB should hold VPNs and the corresponding PFNs. However, the PFN is only necessary when a translation needs to take place. The point of TLB measuring is to simulate the TLB and provide statistics for it, not replicate it. Thus, the only interesting facts for the simulation are whether the TLB resulted in a hit or in a miss. Therefore, the TLB should only contain VPNs.

   The TLB itself can be conceptualised in various ways. Either as a one or two dimensional table. The former, represents the TLB as an array whose size is equal to the number of entries it contains. In case of a fully associative TLB, the whole array is looked up and any entry can be replaced. In case of a direct mapped only one entry can replaced, while, in case of set associative, only one entry within a set can be replaced. A two dimensional representation has a number of rows equal to the number of sets and a number of columns equal to the number of the associativity ways. In a fully associative, there is only one set and the number of ways would equal the number of blocks, while in the case of a direct mapped cache, there is a single column [18]. Figure 4.7 illustrates how is the TLB represented in either case, assuming a 4 way set associative cache with 12 entries. Letters A-L are used to represent where corresponding entries in the 1D and 2D representations should be placed.

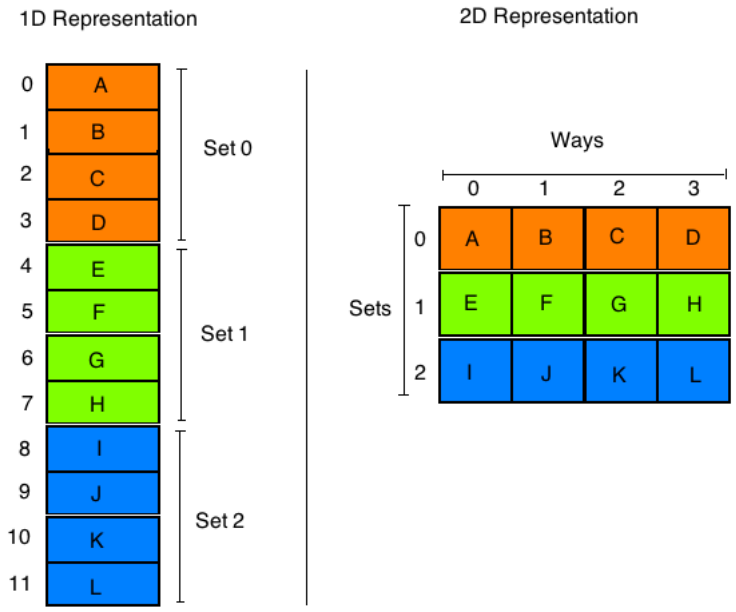

Figure 4.7: 1D and 2D representations of a 4-way associative TLB

The one dimensional representation was chosen, since it was decided that it is easier to

be implemented. Before explaining more about how is the 1D TLB represented and the structures underneath it, TLB associativity should be covered.

At this point the reader is encouraged to recall associativity described in the "*Literature Review*" chapter. Given a virtual address from the CPU, we need to index the cache. This means, to select the set in which the address can be cached [18]. This is done by splitting VPN into two components: *tag* and *index*. The tag represents the address which should be looked up, while the index tells where in the TLB to look for that address. The upper bits of the VPN represent the tag, while the lower bits represent the index. "If the TLB has $T = 2^t$ sets, then the TLB index consists of the t least significant bits of the VPN, and the TLB tag consists of the remaining bits in the VPN" [**cspp**]. In case of a fully associative cache, the index is 0 bits long, since there is only one available set. Thus, in that case, tag=VPN. Moreover, in case of a direct mapped cache, if the width of the index is K bits, it holds that the TLB has 2^K entries. Figure 4.8 represents how is an n sized virtual address with P sized VPN split into the tag, the index and the offset (shown as VPO).

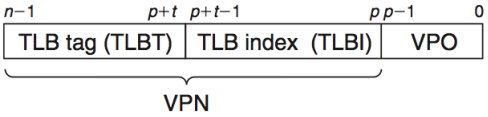

Figure 4.8: Virtual Address split into tag, index and offset [4]

Now that associativity has been explained, we may proceed on explaining the structures underneath the TLB. First of all, a structure for each TLB entry is needed. Each TLB entry should contain a tag and a count field to count the times that this tag has been referenced and resulted in a TLB hit. The count field is only there to help LFU and LRU replacement algorithms. A TLB entry in terms of C code can be represented as follows:

```
1 typedef struct _tlb_entry{
2     Addr tag;
3     Int count;
4 }TLB_ENTRY;
```

Moreover, a TLB can be conceptualised as an array of TLB entries, whose size is equal to the number of entries (type TLB_ENTRY). However, since information about the number of entries is not pre-determined, the TLB has to be allocated dynamically using Valgrind's version of malloc (`VG_(malloc)`). Furthermore, at the time of writing, 3 TLBs exist, an iTLB, a dTLB and a L2TLB, thus 3 TLB variables should exist, one for each TLB. However, creating 3 separate variables to represent each TLB is not good software engineering[6]. In the future more TLBs might be added, so creating a new variable for each of them is not efficient. Instead an array of TLB arrays was created. That way, indexing suffices to access each of the TLBs. The TLB in terms of C code, is represented and initialised as follows:

```
1 TLB_ENTRY **TLB;
2 TLB[0]=(TLB_ENTRY *)VG_(malloc)("itlb", iTLB_entries*sizeof(↩
      TLB_ENTRY));
3 TLB[1]=(TLB_ENTRY *)VG_(malloc)("dtlb", dTLB_entries*sizeof(↩
      TLB_ENTRY));
4 TLB[2]=(TLB_ENTRY *)VG_(malloc)("l2tlb", L2TLB_entries*sizeof(↩
      TLB_ENTRY));
```

[6]Note that the part of Cachegrind that handles the simulation of the CPU caches has separate variables for each of the caches. This is not considered good software engineering since it is not efficient and does not allow for extensions to take place easily.

The difference between "normal" malloc and VG_(malloc) is that the latter requires a string that is used to identify the allocation point [20] and is used to help heap profiling, however, this is beyond the scope of this project.

3. TLB Characteristics

TLB characteristics refer to all kind of information that concern a TLB. This includes the number of hits and misses, the characteristics that define the TLB, information that is crucial for processing virtual addresses, information for page tracking and other type of information.

Characteristics that define the TLB are the degree of associativity and the number of entries. Regarding the degree of associativity, since any number greater than 0 could represent an N-way set associative scheme, it was decided that the value -1 would represent fully associative TLBs, while 0 direct mapped TLBs.

Information that is crucial for processing virtual addresses includes the page size (in bytes), masks for the offset, the VPN, the index and the tag, as well as the total number of sets.

Paging information includes some variables to keep track of all the pages that have entered each TLB and the times each page has been accessed. Tracking pages makes use of a different structure called PAGES which will be explained later.

TLB characteristics in terms of C code are represented as follows with comments explaining each variable:

```
1  /* Holds TLB characteristics */
2  typedef struct _tlb_t{
3
4      //Holds hits and misses per TLB
5      Int hit,miss;
6
7      //Page size should always be in bytes.
8      //i.e. if page size is 4KB then PAGE_SIZE=4*1024=4096
9      ULong page_size;
10     /* assoc represents associativity.
11      * -1 is Fully Associative
12      *  0 is Direct Mapped
13      *  N where N>0 is N-way Associative */
14     Int assoc;
15     //Number of entries
16     Int entries;
17
18     ULong    offset_mask;    //Used to separate Virtual Address ↩
           Offset
19     ULong    vpn_mask;       //Used to separate Virtual Address ↩
           VPN
20     ULong    index_mask;     //used in direct map, to split VPN ↩
           to tag and index.
21     ULong    tag_mask;       //Used in direct map, to get the tag↩
            from VPN.
22     Int      sets;           //Used in N-way assoc, to hold the ↩
           number of sets
23
24     Bool     replace;        //True if TLB was full and from now ↩
           on a replacing algorithm has to choose what to replace. ↩
           Used in FA.
25
26     PAGES    *page_ptr;
27     Int      total_pages;
28
29     Int      tlb_counter;    //Points to which entry is currently↩
            the TLB operating
```

```
30
31      Char    desc_line[128];  //Holds the tlb configuration in ↩
            words.
32  }tlb_t;
```

Again, instead of creating 3 separate variables to hold the information of each of the TLBs, an array of type tlb_t and size 3 was defined. The reason behind the array use is to improve efficiency and allow extensions to be implemented easier. The array was defined as follows:

```
static tlb_t TLBc[3];
```

4. TLB Simulation Algorithm

Since the necessary requirements to measure the TLB (i.e. hold hits/misses, create a structure to simulate TLB, and hold the characteristic of each TLB) have been explained, we proceed on explaining how the simulator works.

First of all, the basic characteristics of each of the TLBs are set, i.e. the degree of associativity and the total number of entries. This is done either by using the CPUID instruction that auto detects each TLB's basic configurations or by allowing the user to manually set those values. File cg-x86-amd64.c, which contains the code to determine hardware characteristics using the CPUID instruction was extended to extract information for each of the TLBs, only if they are implemented. TLB configurations were set according to Intel's Software Developer Manual [6]. TLBs at this stage are represented as cache_t variables. Recall that cache_t contains the following variables: size, assoc and line_size. It was decided that when representing TLBs outside of cg_tlb.c, where cache_t structure is used, size would be treated as the page size, assoc as associativity and line_size would represent the number of entries. A new structure could have been created with the correct variable names, but this "trick" saves more time and does not alter Cachegrind's current structure. Thus, each TLB is represented as a triple {page_size, assoc, entries}. In case that a TLB among iTLB, dTLB and L2TLB was not found the triple {-1,-1,-1} is returned, which corresponds to an invalid TLB that will not be simulated.

A very important fact and drawback of the extension should be noted at this point. The page size is not accessible in user-mode. Specifically, on Intel x86 the page size is determined by a combination of flag values and Control Register (CR) values [6]. Quoting [5], "the MOV to/from Control Register instruction can be executed when the current privileged level is 0", i.e. in kernel-mode. As a consequence of that, the common value of 4KB were chosen to represent the page size. However, it can be manually changed by the user. Moreover, "TLBs are divided into four groups: instruction TLBs for 4-KByte pages, data TLBs for 4-KByte pages; instruction TLBs for large pages (2-MByte, 4-MByte or 1-GByte pages), and data TLBs for large pages. Processors based on Intel Core micro architectures implement one level of instruction TLB and two levels of data TLB. Intel Core i7 processor provides a second-level unified TLB" [6]. Given the chosen 4KB page size, CPUID instruction only determines configurations valid for 4KB pages, i.e. it does not account for large pages. This is achieved by processing CPUID configurations only relevant to 4KB pages. Recall from the code snippet shown in page 29 and its explanation, that Intel CPUID instruction requires indexing which is achieved with the aid of a switch statement. By creating switch cases only relevant to 4KB pages, the code is able to detect configurations relevant to 4KB pages only. Larger pages and the corresponding TLB configurations must be manually set by the user. After all configurations have taken place, they are passed to the TLB simulator to initialise the characteristics of each TLB, i.e. set the values of the variables in each tlb_t. This includes calculating the number of sets (if set-associative scheme is used), the offset, the VPN, as well as the index and tag masks. Finally, the TLB arrays are allocated.

Secondly, after the TLB characteristics have been set, simulation takes place. As it has been decided, the TLB simulator would get notified for the same memory events that the

CPU caches do. Two changes were made to allow this: a) the functions that informed CPU caches for memory events were extended to also inform the TLB simulator and b) a function was created to act as an intermediary between Cachegrind's main file and the TLB simulator.

The intermediary's function name is reference_address(..), is housed under cg_tlb.c (the simulation file) and is called by Cachegrind's main file during or after instrumentation has taken place. They allow the simulator to get notified for memory events taking place in Cachegrind's main file. These functions take as arguments the virtual addresses of the instructions or data, as well as which TLB does that event concern (iTLB or dTLB). Depending on the concerned TLB, a global variable named TLB_TYPE is set accordingly. This is only used to indicate which TLB should be affected and is used to index the TLB[] and TLBc[] arrays described previously. A global variable was chosen instead of passing as a parameter to all of the functions the concerned TLB, since the latter is more time and memory consuming[7]. Then, the intermediary function calls TLB_simulation(..), which is responsible for simulating TLB, with the virtual address as a parameter.

TLB_simulation is responsible for extracting from the given virtual address the VPN and, subsequently, the tag and the index. Then the tag is looked up in the TLB, by calling tlb_lookup(..) with both the tag and the index as parameters. In case that page tracking is enabled, TLB_simulation calls a function to add the current page in a list, if it does not already exist, or otherwise, increase the times accessed counter. Note that tlb_lookup acts differently depending on whether the TLB is fully associative, direct mapped or set associative.

In case of a fully associative TLB, all entries of the relevant TLB (iTLB or dTLB) are searched and checked to match the tag. If one of them matches, the hit counter for the particular TLB is increased. Otherwise, the miss counter is increased and the tag is looked up in the entire L2 TLB (if it exists). If an L2TLB hit occurs, the L2 hit counter is increased and execution continues. If an L2 TLB miss occurs, the L2 miss counter is increased and the missed tag is entered in both L2 and the relevant L1 TLB based on the chosen replacement algorithm. Thus, L2 TLB is inclusive in terms that some L1 elements may be found in L2. In case that an L2 TLB does not exist and an L1 miss occurs, the tag that caused the miss is entered in the relevant L1 TLB again based on the chosen replacement algorithm.

In case of a direct mapped TLB, the entry at index%tlb_entries is checked to see if the tag resides there. Depending on whether the check resulted in a hit or a miss, the relevant TLB hit or miss counter is increased. In case of a miss, the tag is looked in L2 TLB. If found, the L2 TLB hit counter is increased, otherwise the L2 TLB miss counter is increased. In case of a L2 TLB miss, the tag that caused the miss replaces the entry at position index%tlb_entries, in both L1 and L2 TLB.

Finally, in case of a N-way set associative TLB, the entries in the set index%N are checked to see if the tag resides there. Depending on whether the check resulted in a hit or a miss, the relevant TLB hit or miss counter is increased. In case of a miss, the tag is looked in L2 TLB. If found, the L2 TLB hit counter is increased, otherwise the L2 TLB miss counter is increased. Again, in case of a L2 TLB miss, the tag that caused the miss replaces any entry in the set in both L1 and L2 TLB. Which entry to replace is based on the selected replacement algorithm.

4.3.2 Page Tracking

Besides providing statistical information for the TLB, Cachegrind is now able to keep track of the pages used. This is provided as an option whose default value is off, i.e. the

[7]If the concerned TLB is passed as a parameter in a series of, for instance, 5 function calls, the memory occupied on stack would be $5 * 4 = 20$ bytes (assuming 4 byte sized integers). On the other hand, one global variable no matter the number of function calls will occupy 4 bytes.

user has to specify that he wants page records to be kept. There are two reasons behind this choice. Firstly, page tracking's output is not considered of paramount importance, and secondly, as it will be shown later, it introduces a slowdown which might be annoying to wait for in every execution.

The reason that page tracking was developed is because it was not difficult to be implemented and could be helpful for some users. However, we cannot think of any particular reason that page tracking can be considered useful, to the extend that it can help users increase the cache-friendliness of their programs.

The notion behind page tracking is simple. Store all tags in a dynamic data structure. Before storing a tag check if it already exists, if yes increase the times used, if no add it to the data structure. Since the number of virtual addresses to be generated is not known, a dynamic data structure was chosen. Specifically, a linked list was implemented, where each node (type PAGES) had three variables. One to represent the tag, one to hold the access times and one to hold the next node in the list. In terms of C code PAGES structure is represented as follows:

```
1 typedef struct pages_accessed{
2     Addr tag;  //tag should be put here
3     ULong count; //ULong cause it might get big enough
4     struct pages_accessed *next;
5 }PAGES;
```

Recall that the linked list is stored within tlb_t, since the pages need to be tracked per TLB.

Instead of a linked list which has O(n) searching complexity, the OSet explained in section 3.2.2 could have been used instead which has O(log(n)) complexity or better [20].

4.3.3 Per Source Code Line Hits & Misses

The original version of Cachegrind allows gathering profiling information, which can be used at a later time to provide a detailed presentation of that information [10]. The part of the tool responsible for gathering that information is the main file. The main file, with the aid of the Cost Centre table (CC_table) (described in section 3.2.2), holds profiling information. Recall that the CC_table acts as a database where each element is of type LineCC. LineCC represents every source line instrumented (grouped by file name, function name and line number) and holds information that include the CPU caches. Specifically, LineCC contains 3 CacheCC elements, one for instruction reads (Ir), one for data reads (Dr) and one for data writes (Dw). Each CacheCC holds the total number of each of the Ir, Dr and Dw, as well as their misses in each cache level.

In order to gather profiling information for the TLB, a new cost centre structure had to be created especially for TLB, named TLBCC. CacheCC could have been used instead, but it was decided to have separate cost centre structures for the TLB and the CPU caches. After L2 cache inclusion took place, CacheCC contained 4 elements, 3 of which represent misses in level 1, level 2 and level 3 caches. At the time of writing, two levels of TLB exist, thus an extra variable would unnecessarily occupy memory. Moreover, in case that an extension or changes might take place later regarding the CPU caches or the TLB, it is better to be isolated and not have an effect on each other. Therefore, TLBCC was created containing the same elements as CacheCC but for TLB. These are the total number of accesses of this kind and misses in the first and the second TLB levels. In terms of C code TLBCC is represented as follows:

```
1 typedef
2     struct {
3         ULong a;  /* total # memory accesses of this kind */
4         ULong t1; /* Misses in the first level TLB */
5         ULong t2; /* Misses in the second level TLB */
6 }TLBCC;
```

Three elements of TLBCC were added in LineCC to hold Ir, Dr and Dw for the TLB. In terms of C code the extended LineCC has as follows:

```
1  typedef struct {
2     CodeLoc  loc; /* Source location that these counts pertain to */
3
4     CacheCC  Ir;  /* Insn read counts */
5     CacheCC  Dr;  /* Data read counts */
6     CacheCC  Dw;  /* Data write/modify counts */
7
8     TLBCC    t_Ir; /* TLB insn read counts */
9     TLBCC    t_Dr; /* TLB data read counts */
10    TLBCC    t_Dw; /* TLB data write/modify counts */
11
12    BranchCC Bc;  /* Conditional branch counts */
13    BranchCC Bi;  /* Indirect branch counts */
14 } LineCC;
```

Figure 4.9 illustrates the bigger picture of all the structures related to per source code line profiling.

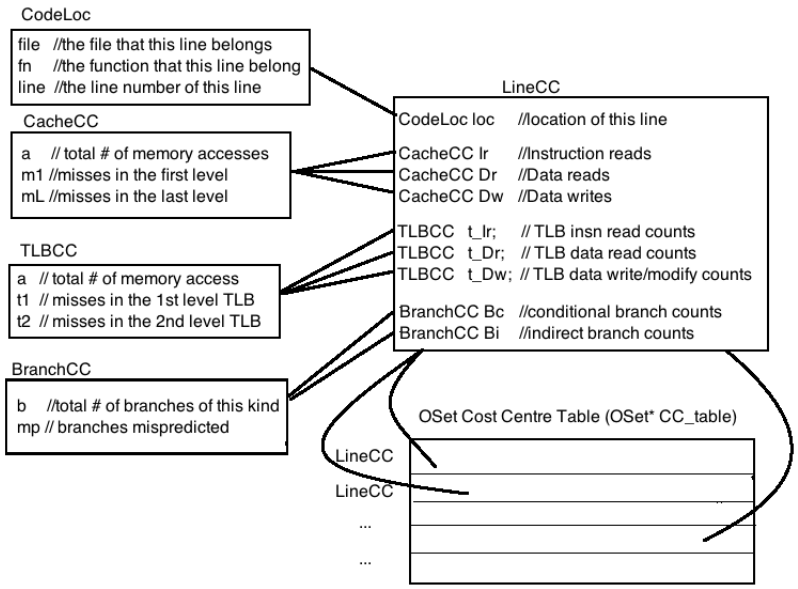

Figure 4.9: Cost Centre Structures

TLBCC misses are updated by the TLB simulator. In the intermediary function, along with the virtual addresses and the relevant TLB that should get updated, a pointer that points in the TLBCC misses is passed as a parameter and gets updated when a miss occurs in the same way that the simulator treats all misses. Finally, Cachegrind's output file was extended to include TLB Ir, Dr and Dw counts, so that annotation could take place afterwords. Note that the annotation program, cg_annotate, had not to change, since it automatically detects the type of information that should include. This is good software engineering, since no matter the extensions made to Cachegrind, if they are in line with the rest of the tool, annotation will work.

4.4 Running the TLB Extension

In this section the TLB extension is executed in order to demonstrate that it works, the output of the extension is explained and the available options are explored. This section does not consider whether results shown are correct or not. The validity of the results is checked and discussed in a next section.

4.4.1 Basic Execution

The basic execution consists of running Cachegrind with no parameters. Cachegrind will print two kinds of information: a)the technical characteristics of the simulated TLBs and b) per TLB statistics regarding the hits and misses.

TLB technical characteristics include TLB associativity, TLB page size and the number of entries a TLB has. Per TLB statistics include the total number that a particular TLB was accessed, the number of hits and misses, as well as hit and miss ratios. Moreover, information that can be used for later analysis and annotation will be written in Cachegrind's generated log file, however, this will be covered later. Cachegrind will auto-detect using CPUID instructions the characteristics of each TLB and will simulate it. The default page size used is 4KB for reasons explained earlier.

To demonstrate the basic execution, the Unix command `ls` will be used. `ls` lists the contents of a directory, however, its output will be cropped since it would occupy a lot of space.

By executing Cachegrind the following results are obtained (analysis will follow):

```
1  $valgrind --tool=cachegrind ls
2  ==3148== Cachegrind, a TLB, cache and branch-prediction profiler
3  ==3148== Copyright (C) 2002-2012, and GNU GPL'd, by Nicholas Nethercote et al.
4  ==3148== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
5  ==3148== Command: ls
6  ==3148==
7  --3148-- warning: L4 cache ignored
8  --3148-- warning: L3 cache found, using its data for the LL simulation.
9  ...(ls command output cropped)...
10 ==3148==
11 ==3148== I   refs:      987,465
12 ==3148== I1  misses:      1,631
13 ==3148== L2i misses:      1,510
14 ==3148== LLi misses:      1,505
15 ==3148== I1  miss rate:    0.16%
16 ==3148== L2i miss rate:    0.15%
17 ==3148== LLi miss rate:    0.15%
18 ==3148==
19 ==3148== D   refs:      494,321  (353,809 rd   + 140,512 wr)
20 ==3148== D1  misses:      4,185  (  3,433 rd   +     752 wr)
21 ==3148== L2d misses:      2,880  (  2,201 rd   +     679 wr)
22 ==3148== LLd misses:      2,826  (  2,163 rd   +     663 wr)
23 ==3148== D1  miss rate:     0.8% (    0.9%     +     0.5%  )
24 ==3148== L2d miss rate:     0.5% (    0.6%     +     0.4%  )
25 ==3148== LLd miss rate:     0.5% (    0.6%     +     0.4%  )
26 ==3148==
27 ==3148== L2 refs:         5,816  (  5,064 rd   +     752 wr)
28 ==3148== L2 misses:       4,390  (  3,711 rd   +     679 wr)
29 ==3148== L2 miss rate:      0.2% (    0.2%     +     0.4%  )
30 ==3148==
31 ==3148== LL refs:         4,390  (  3,711 rd   +     679 wr)
32 ==3148== LL misses:       4,331  (  3,668 rd   +     663 wr)
33 ==3148== LL miss rate:      0.2% (    0.2%     +     0.4%  )
34 ==3148==
35 ==3148==
36 ==3148==
37 ==3148== ---TLB characteristics---
38 ==3148== Virtual Address Size:     32 bits
39 ==3148== Replacement Policy:       Least Recently Used
40 ==3148==
41 ==3148== TLB type:          iTLB  (L1 Instruction TLB)
42 ==3148== Associativity:     8-Way Associative
43 ==3148== Page Size:         4096 bytes
44 ==3148== Entries:           64
```

```
45 ==3148==
46 ==3148==
47 ==3148== TLB type:           dTLB  (L1 Data TLB)
48 ==3148== Associativity:     4-Way Associative
49 ==3148== Page Size:         4096 bytes
50 ==3148== Entries:           64
51 ==3148==
52 ==3148==
53 ==3148== TLB type:           L2TLB (L2 Unified TLB)
54 ==3148== Associativity:     8-Way Associative
55 ==3148== Page Size:         4096 bytes
56 ==3148== Entries:           1024
57 ==3148==
58 ==3148==
59 ==3148==
60 ==3148== ---Results---
61 ==3148==
62 ==3148== ---iTLB Stats---
63 ==3148== Total Accesses:   987465
64 ==3148== Hits:             987315
65 ==3148== Misses:           150
66 ==3148== Hit ratio:        99.9%
67 ==3148== Miss ratio:       0.0%
68 ==3148==
69 ==3148==
70 ==3148== ---dTLB Stats---
71 ==3148== Total Accesses:   494321
72 ==3148== Hits:             493934
73 ==3148== Misses:           387
74 ==3148== Hit ratio:        99.9%
75 ==3148== Miss ratio:       0.0%
76 ==3148==
77 ==3148==
78 ==3148== ---L2TLB Stats---
79 ==3148== Total Accesses:   537
80 ==3148== Hits:             313
81 ==3148== Misses:           224
82 ==3148== Hit ratio:        58.2%
83 ==3148== Miss ratio:       41.7%
84 ==3148==
85 ==3148==
```

Warning Message Analysis

The generated warning messages in lines 7 and 8 are almost the same with those generated in "*Chapter 3 — L2 Cache Inclusion*" in section "*Warning Message Analysis*", and their explanation can be found there. The only difference is that at that section, Cachegrind had generated a warning related to a TLB configuration (0x63 in particular), which now, due to the TLB extension is no longer present.

Consistency of Results

Results may be slightly different from execution to execution, but the difference is considered negligible and results can still be helpful to the users. This issue is also present in the original Cachegrind and is not a side effect of the extensions. It occurs due to the way that Cachegrind handles memory events, and since the TLB extension uses the same way to handle memory events, the consistency issue is inherited.

Precision Error

As it may have been noticed, hit and miss ratios if added together, they do not equal 100% which is the expected outcome, but they are equal to 99.9%. The 0.1% error is not a side effect of the extensions and is also present in the original Cachegrind.

TLB Characteristics

After CPU cache output stops and before statistics gathered per TLB are shown, the characteristics of each TLB are presented (lines 37-56). As it can be seen, the size of the virtual address space used is 32 bits with LRU replacement policy. Moreover, the simulated TLBs are the following:

- An 8-way associative iTLB with 4KB pages and 64 entries.
- A 4-way associative dTLB with 4KB pages and 64 entries.
- An 8-way associative second level (L2) unified TLB with 4KB pages and 1024 entries.

TLB Statistics & Interpretation

At the end of the execution output (lines 60-85), per TLB results are shown. A header "—XTLB Stats—", where X represents a particular TLB, is used to indicate where each TLB's results begin. Then, the total number of accesses for that TLB is shown, among with the number of hits, misses, hit ratio and miss ratio.

As it can be seen, the number of total accesses of iTLB and dTLB equal the number of I1 and D1 references respectively. This is expected, since as described at the beginning of section "*4.6 — Design & Implementation*", the same instructions that Cachegrind sends in I1 cache are sent in iTLB and the same data that Cachegrind sends in D1 is sent in dTLB.

Moreover, the hit ratios of L1 TLBs (iTLB and dTLB) are high, close to 100%. This is also as expected, since the normal TLB hit ratio is close to 99% [23]. On the other hand, L2 LTB is not that high (58.2%), however, the number of L2TLB total accesses is very low and does not allow for L2TLB to exploit spatial and temporal locality. Probably tags that caused a miss are "dispersed" and not tightly packed in memory. Moreover, the missed tags are probably not quickly re-referenced in time.

By interpreting the TLB results for the executed program, `ls`, we conclude that it is TLB-friendly. Most of the virtual to physical address translations are completed using the TLB, and only 224 (L2 misses) access the page table stored in main memory. `ls` has probably been developed in a way that spatial and temporal locality are used efficiently. Data accessed should be packed together in memory and may be probably re-referenced in short time periods. Moreover, L2TLB saved memory access time, since it grasped most of the misses produced by both iTLB and dTLB (58.2% (313 out of 537)).

4.4.2 Page Tracking

Cachegrind is also able to keep track of all the pages used and the times used per TLB. To enable page tracking the command line option "`--tlb-page-sim=yes`" has to be entered. The results of this option will be printed right after TLB statistics are shown.

In order to demonstrate page tracking, `ls` will be used again. Since the output regarding the CPU caches and the TLB is the same as before, it will not be shown. Moreover, not all pages for each TLB will be shown since they would occupy a lot of space. Only a portion of the pages accessed will be shown in order to familiarise the reader with the program's output.

```
1 $valgrind --tool=cachegrind --tlb-page-sim=yes ls
2 ...(Cachegrind's output cropped)..
3 ==14046== ---Pages Accessed---
4 ==14046==
5 ==14046== iTLB Pages Accesed
6 ==14046== Pages Accessed In total:   36
```

```
==14046== 1) Page 00000830, accessed 1118 times
==14046== 2) Page 00000822, accessed 146 times
.....(page tracking results not shown)....
==14046== 34) Page 00000801, accessed 237529 times
==14046== 35) Page 00000802, accessed 73013 times
==14046== 36) Page 00000800, accessed 14996 times
==14046==
==14046== dTLB Pages Accesed
==14046== Pages Accessed In total:   25
==14046== 1) Page 0000bec7, accessed 25284 times
==14046== 2) Page 00000435, accessed 1 times
.....(page tracking results not shown)....
==14046== 23) Page 00000400, accessed 1312 times
==14046== 24) Page 00000401, accessed 8445 times
==14046== 25) Page 0000bec8, accessed 200037 times
==14046==
==14046== L2TLB Pages Accesed
==14046== Pages Accessed In total:   10
==14046== 1) Page 000017d8, accessed 1 times
==14046== 2) Page 00000086, accessed 1 times
.....(page tracking results not shown)....
==14046== 8) Page 00000100, accessed 4619 times
==14046== 9) Page 000017d9, accessed 2 times
==14046== 10) Page 00000080, accessed 10974 times
```

4.4.3 Exploring Program Options

An explanation of all of the available options as well as how can they be used is described within this section. No executions will take place nor any output will be shown. This section only focuses on just explaining how to use each option and what does it do. Note that the right place to enter options is between the tool name (*cachegrind* in this case) and the name of the executed program (*prog*):
`valgrind --tool=cachegrind (options) prog`

Measure TLBs

This option is by default on and TLBs are measured every time the extended version of Cachegrind is executed. If desired it may be disabled as follows:
`--tlb-sim=no`
The values this option can take and the default chosen value (in square brackets) can be seen here:
`--tlb-sim=yes\no    [yes]`

Track Pages

As it has already been explained, this option is disabled by default. To enable it, one may enter the following option:
`--tlb-page-sim=yes`

The values this option can take and the default chosen value (in square brackets) can be seen here:
`--tlb-sim=yes\no    [no]`

Set TLBs Manually

The user might want to manually set the characteristics of each TLB either because he wants to observe a program in a specific TLB environment or because he might want to use larger page sizes that cannot be auto-detected. In either case the user is able to specify each TLB as follows:

- `--iTLB=<page_size>,<assoc>,<entries>`

- `--dTLB=<page_size>,<assoc>,<entries>`
- `--L2TLB=<page_size>,<assoc>,<entries>`

No default values exist for the TLBs, since they are auto-detected. If a TLB cannot be detected, then it is assumed that it is not installed and therefore it is not simulated.

Set Virtual Address Space Size

Since the size of the virtual addresses is not auto-detected and may not be the one desired by the user an option allows to be manually set:
`--tlb-vas-size=<num>`
Where *num* can be any number bigger than 0. Although, virtual address sizes should be a power of 2 [**cspp**], most of the times, when simulating 64 bit addresses, the OS actually uses 48-bit addresses which is not a power of 2. Therefore, in order to allow this kind of simulation no other checking is done in terms of what is accepted as size. The user is responsible for setting this correctly.

The default value set for the virtual address space size is 32 bits.

Set Replacement Policy

The user is able to choose among three replacement policies: LFU, LRU and Random. One might set the replacement policy he desires as follows:
`--tlb-rep-pol=<num>`
Where *num* can take one of three values. 0 corresponds to LFU, 1 corresponds to LRU and 2 corresponds to Random. The default replacement policy used is LRU since it is most likely to be used by a TLB. Any other value entered besides those mentioned will result in using LRU.

4.4.4 Per Source Code Line Information

Cachegrind now provides per file, per function and per source code line information for the number of TLB accesses and misses of Ir, Dr, and Dw. That kind of information is not available in basic execution and can only be accessed by later analysis and annotation of Cachegrind's log files. As it was stated before, later analysis and annotation is achieved by running cg_annotate on a generated log file. cg_annotate is a product of good software engineering, since it automatically detects the amount of information written in the log files. Thus, cg_annotate is able to detect TLB data and present it properly, without the need to extend its code. Cachegrind collects information for a number of memory events regarding CPU caches and TLB. cg_annotate, unless specified otherwise, will show all of them. However, for the following demonstration only the information related to TLB will be shown.

In order to demonstrate per source code line statistics two things are required: a) to have the executed program's source code and b) to compile the program with extra debugging information. Using GCC, the latter is achieved by using the `-g` flag. Since `ls`'s program code large (5,000 lines), for the purposes of this demonstration a simple C program will be used. Assume a C program, named array.c, that creates a 100 sized integer array and sets each cell to some value:

Listing 4.1: array.c

```
1  #define SIZE (100)
2  int main(){
3
4              int i,array[SIZE];
5
6              for(i=0;i<SIZE;i++){
7                 array[i]=i+10;
8              }
9
10           return 0;
11 }
```

By compiling array.c with `-g` flag enabled as follows:
`gcc -g -o array array.c`
and then by executing Cachegrind upon it, the following results are obtained related to TLB gathered in a table:

	iTLB	**dTLB**	**L2TLB**
Virtual Address Space Size	32	32	32
Replacement Policy	LRU	LRU	LRU
Associativity	8	4	8
Page Size	4096 bytes	4096 bytes	4096 bytes
Entries	64	64	1024
Total Accesses	109594	51037	89
Hits	109557	50985	4
Misses	37	52	85
Hit Ratio	99.9%	99.8%	4.4%
Miss Ratio	0.0%	0.1%	95.5%

A log file named *cachegrind.out.2780* was generated in the folder that the execution took place. To execute cg_annotate upon this log file, results need to be limited to be TLB related and the absolute path of array.c must be given. This is done as follows:

```
$cg_annotate cachegrind.out.2780 --show=TIr,TI1mr,TI2mr,TDr,TD1mr,↩
    TD2mr,TDw,TD1mw,TD2mw ~/Desktop/array.c
```

and results to this which is explained right after:

```
--------------------------------------------------------------------------------
I1 cache:         32768 B, 64 B, 8-way associative
D1 cache:         32768 B, 64 B, 8-way associative
L2 cache:         262144 B, 64 B, 8-way associative
LL cache:         6291456 B, 64 B, 12-way associative
iTLB cache:         4096 B, 64 E, 8-wayAssociative
dTLB cache:         4096 B, 64 E, 4-wayAssociative
L2TLB cache:         4096 B, 1024 E, 8-wayAssociative
Command:          ./array
Data file:        cachegrind.out.2780
Events recorded:  Ir I1mr I2mr ILmr TIr TI1mr TI2mr Dr D1mr D2mr DLmr
                  TDr TD1mr TD2mr Dw D1mw D2mw DLmw TDw TD1mw TD2mw
Events shown:     TIr TI1mr TI2mr TDr TD1mr TD2mr TDw TD1mw TD2mw
Event sort order: Ir I1mr I2mr ILmr TIr TI1mr TI2mr Dr D1mr D2mr DLmr
                  TDr TD1mr TD2mr Dw D1mw D2mw DLmw TDw TD1mw TD2mw
Thresholds:       0.1 100 100 100 100 100 100 100 100 100 100 100 100
                  100 100 100 100 100 100 100 100
Include dirs:
User annotated:   /home/(..)/Desktop/array.c
Auto-annotation:  off

--------------------------------------------------------------------------------
    TIr TI1mr TI2mr    TDr TD1mr TD2mr    TDw TD1mw TD2mw
--------------------------------------------------------------------------------
109,594    37    34 36,299    42    41 14,738    10    10  PROGRAM TOTALS

--------------------------------------------------------------------------------
    TIr TI1mr TI2mr    TDr TD1mr TD2mr    TDw TD1mw TD2mw  file:function
--------------------------------------------------------------------------------
104,063    32    29 34,579    42    41 13,338    10    10  ???:???
```

```
31    3,492      1      1    964      0      0    927      0      0  ???:bsearch
32      710      0      0    403      0      0    102      0      0  /(..)/Desktop/array.c:main
33      630      0      0    159      0      0    150      0      0  ???:__libc_memalign
34      198      0      0     54      0      0     90      0      0  ???:malloc
35      168      0      0     48      0      0     48      0      0  ???:calloc
36
37 --------------------------------------------------------------------------------
38 -- User-annotated source: /home/(..)/Desktop/array.c
39 --------------------------------------------------------------------------------
40 TIr  TI1mr  TI2mr  TDr  TD1mr  TD2mr  TDw  TD1mw  TD2mw
41
42   .      .      .    .      .      .    .      .      .
43   .      .      .    .      .      .    .      .      .  #define SIZE (100)
44   3      0      0    0      0      0    1      0      0  int main(){
45   .      .      .    .      .      .    .      .      .
46   .      .      .    .      .      .    .      .      .            int i,array[SIZE];
47   .      .      .    .      .      .    .      .      .
48 304      0      0  201      0      0    1      0      0            for(i=0;i<SIZE;i++){
49 400      0      0  200      0      0  100      0      0               array[i]=i+10;
50   .      .      .    .      .      .    .      .      .            }
51   .      .      .    .      .      .    .      .      .
52   1      0      0    0      0      0    0      0      0         return 0;
53   2      0      0    2      0      0    0      0      0  }
54
55 --------------------------------------------------------------------------------
56 TIr  TI1mr  TI2mr  TDr  TD1mr  TD2mr  TDw  TD1mw  TD2mw
57 --------------------------------------------------------------------------------
58   1      0      0    1      0      0    1      0      0  percentage of events annotated
```

Results were limited to show only the total number of TLB Irs (TIr)[8], the number of Ir misses in L1 TLB (TI1mr) and L2 TLB (TI2mr), the total number of TLB Drs (TDr), the number of Dr misses in L1 and L2 TLB (TD1mr and TD2mr respectively), the number of total TLB Dw (TDw), as well as the number of Dw misses in L1 and L2 TLB (TD1mw and TD2mw respectively).

Lines 1-20 show the characteristics of both CPU caches and the TLB, the command by which Cachegrind was executed (./array.c), the log's file name (cachegrind.out.2780), as well as all the events recorded and which of these are shown in this execution. "*Events recorded*", "*Event sort order*" and "*Thresholds*" were edited to occupy two lines for appearance reasons, while real execution shows them in one line. Moreover, it should be noted that paths to files were edited using "(..)" for appearance reasons again.

Lines 22-26 show the overall statistics of the TLB related memory events of the program. Linking the results shown in line 25 with those gathered by the basic execution and were presented in the table at page 56 it can be seen that the total number of TLB Ir (TIr) equals the number of iTLB total accesses (both 109,594), and the number of TLB Ir level 1 misses (TI1mr) equals the number of iTLB misses (both 37). Moreover, the number of TLB Dr (TDr) and TLB Dw (TDw) added together (36,299 + 14,738 = 51,037) equal the number of dTLB total accesses, the number of TLB Dr and TLB Dw misses in level 1 (TD1mr, TD1mw) (42 + 10 = 52) equal the total number of dTLB misses, the number of misses in level 1, i.e. TI1mr, TD1mr, TD1mw (37 + 42 + 10 = 89), equal the number of total L2TLB accesses, and finally the misses in level 2 (TI2mr, TD2mr, TD2mw (34 + 41 + 10 = 85)) equal the number of L2TLB misses.

Lines 27-36 show the global and function-level counts which include file and function name identifications. However, some files and functions names could not be detected and thus are represented by "???". This issue is also present in original Cachegrind and is not a side effect of the extension. Results shown indicate that it is not array.c that causes most of TLB misses, but instead it is a function in a file that cannot be identified. That function is probably a library function since libraries were not compiled using -g and thus, cannot be

[8]The capital T in front of each memory event indicates that it is related to TLB.

identified.

Lines 37-54 show the per source code line statistics. The particular program does not cause any misses. Columns that contain a dot indicate that an event cannot happen. "This is useful for distinguishing between an event which cannot happen, and one which can but did not" [10].

Finally, lines 55-58 indicate the percentage of the events annotated.

4.5 Validity of Results

The current section focuses on checking whether TLB measuring results are correct or not. Unfortunately, there is not a simple and direct way to check upon the validity of the results as it was the case in L2 cache inclusion in section "*3.6 — Validity of Results*". The method devised to draw upon the validity of the results is to execute the TLB extension with both TLB-friendly and TLB-unfriendly programs where the number of misses can be predicted and observe if results are along the lines of the expectations.

Although, since it is difficult to create a general TLB-friendly or TLB-unfriendly program that would work for both instructions and data, separate programs have been devised for each TLB. Recall that the underlined TLB characteristics of the computer that executions take place are the following:

- An 8-way associative iTLB with 4KB pages and 64 entries.
- A 4-way associative dTLB with 4KB pages and 64 entries.
- An 8-way associative second level (L2) unified TLB with 4KB pages and 1024 entries.

Following programs will be created with the above TLB characteristics in mind.

Also, recall that TLB is based on the phenomena of spatial and temporal locality. Within the TLB context, spatial locality refers to accessing elements that are tightly packed in memory, while temporal locality refers to quick re-referencing of elements in time. A TLB-friendly program would be one that is along the lines of the above two ideas and does not deviate, while a TLB-unfriendly program is exactly the opposite. A TLB-unfriendly program should be subject to inefficient spatial locality and to limited or no temporal locality. The former refers to accessing elements that are not in the same page, while the later refers to not accessing recently accessed elements.

4.5.1 Checking dTLB

dTLB Unfriendly

dTLB is considered first, starting with a dTLB-unfriendly program. A data inefficient program would be one that creates an array and accesses its elements only once in an inefficient way resulting in many TLB misses. Thus, the effect of spatial locality would be minimised, while the concept of temporal locality would be of no use. dtlb-unfriendly.c is such a program whose code can be seen below, while its analysis follows afterwards:

```
1  #define ENTRIES (3000)
2  #define SIZE (1024)
3
4  int main(void){
5
6          int x,y;
7          static int a[ENTRIES][SIZE];
8
9
10         for(x=0;x<SIZE;x++)
11                 for(y=0;y<ENTRIES;y++)
12                     a[y][x]=0; //column-major traversal
13
```

```
14          return 0;
15  }
```

dtlb-unfriendly.c creates an integer array of 3,072,000 integers. Recall from section "*4.1.4 — TLB Operation Example"* that array items are packed next to each other in memory in order from 0 to N-1 assuming a N-sized array. A common configuration of 4KB page size with 4 byte sized integers allows a total of 1024 (=4096/4) integers to be placed per page. If array items were accessed in a cache friendly way, a TLB miss would be expected every 1024 integer accesses i.e. every time a new page had to be accessed. However, dtlb-unfriendly.c performs a column-major traversal: it accesses array's elements in a way that every access is in a different page, so every access is expected to cause a miss. Thus, a total number of ENTRIES * SIZE (i.e. 3000 * 1024) misses are expected, which in this case are 3,072,000 misses. Figure 4.10 illustrates how is the array placed in memory and the contents of each page.

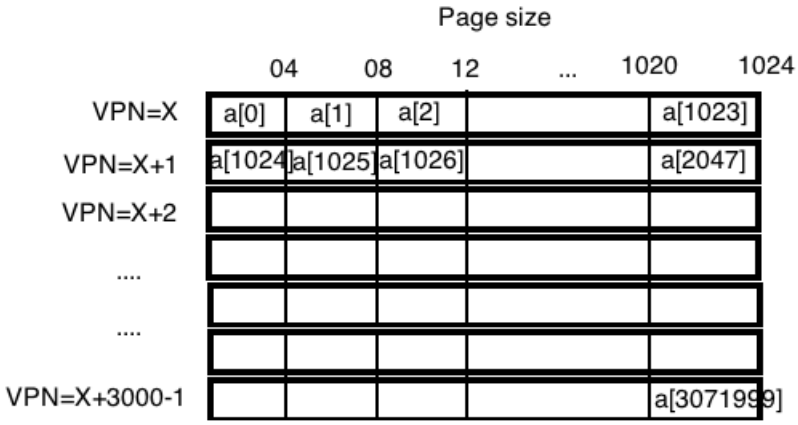

Figure 4.10: A 3000*1024 array placed in memory

By executing the TLB extension with dtlb-unfriendly.c we expect a number of 3,072,000 misses regarding the dTLB. Moreover, information regarding iTLB and L2TLB in this case is of no importance, thus they are manually deactivated.

```
1  $gcc -g -o dtu dtlb-unfriendly.c
2  $valgrind --tool=cachegrind --iTLB=-1,-1,-1 --L2TLB=-1,-1,-1 ./dtu
3  ...(Cachegrind output cropped)...
4  ==20214== ---Results---
5  ==20214==
6  ==20214== ---dTLB Stats---
7  ==20214== Total Accesses:   15414633
8  ==20214== Hits:             12342561
9  ==20214== Misses:           3072072
10 ==20214== Hit ratio:        80.0%
11 ==20214== Miss ratio:       19.9%
```

As it can be seen in line 9, the number of misses (3,072,072) is very close to the expected number (3,072,000). A more detailed analysis can take place using cg_annotate:

```
1  $cg_annotate cachegrind.out.20214 --show=TIr,TI1mr,TDr,TD1mr,TDw,TD1mw ~/Desktop
2  /dtlb-unfriendly.c
3
4  ...(cg_annotate output cropped)...
5  --------------------------------------------------------------------------------
6  -- User-annotated source: /home/(..)/Desktop/dtlb-unfriendly.c
7  --------------------------------------------------------------------------------
```

```
8          TIr  TI1mr          TDr  TD1mr          TDw       TD1mw
9
10           .      .            .      .            .           .   #define ENTRIES (3000)
11           .      .            .      .            .           .   #define SIZE (1024)
12           .      .            .      .            .           .
13           3      0            0      0            1           0   int main(void){
14           .      .            .      .            .           .
15           .      .            .      .            .           .     int x,y;
16           .      .            .      .            .           .     static int a[ENTRIES][SIZE];
17           .      .            .      .            .           .
18           .      .            .      .            .           .
19       3,076      0        2,049      0            1           0     for(x=0;x<SIZE;x++)
20   9,220,096      0    6,145,024      0        1,024           0             for(y=0;y<ENTRIES;y++)
21  12,288,000      0    6,144,000      0    3,072,000   3,071,999             a[y][x]=0;
22           .      .            .      .            .           .
23           1      0            0      0            0           0     return 0;
24           2      0            2      0            0           0   }
```

As it can be seen by the number of DTLB data write misses (TD1mw) the number of misses caused by accessing the array (code line: *a[y][x]=0;*) and setting each value is equal to 3,071,999, which differs only by one from the expected result (3,072,072). Moreover, the total number of dTLB data write accesses (TDw) in that line equals the number of array elements (3,072,000) which is also as expected, since the memory is written once for every element in the array.

dTLB Friendly

We now consider the dTLB friendly code, which performs a row-major traversal which is expected to cause a miss every 1024 array accesses, i.e. a miss for every new page access. Therefore, the total number of misses would be equal to the total number of TLB entries, which, in our case, are determined by the size of ENTRIES (3000). dtlb-friendly.c differs from dtlb-unfriendly.c on how is the array accessed. Its code is the following:

Listing 4.2: dtlb-friendly.c

```
1  #define ENTRIES (3000)
2  #define SIZE (1024)
3
4  int main(void){
5
6          int x,y;
7          static int a[ENTRIES][SIZE];
8
9
10         for(x=0;x<ENTRIES;x++)
11                 for(y=0;y<SIZE;y++)
12                 a[x][y]=0; //row-major traversal
13
14         return 0;
15 }
```

By compiling and executing dtlb-friendly.c on Cachegrind the following results are obtained:

```
1  $gcc -g -o dtf dtlb-friendly.c
2  $valgrind --tool=cachegrind --iTLB=-1,-1,-1 --L2TLB=-1,-1,-1 ./dtf
3  ...(Cachegrind output cropped)...
4  ==20366== ---Results---
5  ==20366==
6  ==20366== ---dTLB Stats---
7  ==20366== Total Accesses:   15422537
8  ==20366== Hits:             15419464
9  ==20366== Misses:           3073
```

```
10 ==20366== Hit ratio:         99.9%
11 ==20366== Miss ratio:        0.0%
```

As it can be seen in line 9, the number of dTLB misses is 3,073, which is close to the expected outcome (3,000). Again, a more detailed analysis can take place using cg_annotate:

```
1  ...(cg_annotate output cropped)...
2  --------------------------------------------------------------------------------
3  -- User-annotated source: /(..)/Desktop/dtlb-friendly.c
4  --------------------------------------------------------------------------------
5          TIr  TI1mr          TDr  TD1mr          TDw  TD1mw
6
7            .      .            .      .            .      .  #define ENTRIES (3000)
8            .      .            .      .            .      .  #define SIZE (1024)
9            .      .            .      .            .      .
10           3      0            0      0            1      0  int main(void){
11           .      .            .      .            .      .
12           .      .            .      .            .      .          int x,y;
13           .      .            .      .            .      .          static int a[ENTRIES][SIZE];
14           .      .            .      .            .      .
15           .      .            .      .            .      .
16       9,004      0        6,001      0            1      0          for(x=0;x<ENTRIES;x++)
17   9,228,000      0    6,147,000      0        3,000      0                  for(y=0;y<SIZE;y++)
18  12,288,000      0    6,144,000      0    3,072,000  3,000                  a[x][y]=0;
19           .      .            .      .            .      .
20           1      0            0      0            0      0          return 0;
21           2      0            2      0            0      0  }
```

As TD1mw column results indicate, the number of misses for the line that accesses the array is 3,000 which is exactly the expected value. Moreover, the total number of dTLB data write accesses (TDw) in that line equals the number of array elements (3,072,000) which is also as expected for reasons described before.

Based on the results obtained by dtlb-unfriendly.c and dtlb-friendly.c it is concluded that dTLB results can be considered valid and the way dTLB measuring is structured is correct.

4.5.2 Checking iTLB

iTLB Friendly code

Besides dTLB, iTLB needs to be checked. Although, the rationale on how to check iTLB is the same as dTLB (by using iTLB friendly and unfriendly programs), iTLB cannot be checked using the exact same programs used for dTLB checking, since these programs were based on the phenomenon of spatial and temporal locality regarding data and not instructions. Thus, two programs need to be developed: one that is iTLB friendly and one that is iTLB unfriendly. The former is easier to be developed as it simply requires instructions that are close together in memory and are quickly re-referenced in time so that their entries in TLB are not overwritten by other instructions and thus, accessing them leads to a hit. Such a program could be a simple loop that loops 1000 times and prints numbers from 0 to 999. The code of such a program called itlb-friendly.c can be seen below:

Listing 4.3: itlb-friendly.c

```
1 #include <stdio.h>
2
3 int main(){
4
5         int i;
6         for(i=0;i<1000;i++)
7                 printf("%d\n",i);
8
9 }
```

By compiling it and disassembling it using GDB we obtain the following:

```
1  $gcc -g -o itf itlb-friendly.c
2  $gdb -q ./itf
3  Reading symbols from /home/(..)/Desktop/itf...done.
4  (gdb) set disassembly intel
5  (gdb) disassemble main
6  Dump of assembler code for function main:
7     0x080483e4 <+0>:     push   ebp
8     0x080483e5 <+1>:     mov    ebp,esp
9     0x080483e7 <+3>:     and    esp,0xfffffff0
10    0x080483ea <+6>:     sub    esp,0x20
11    0x080483ed <+9>:     mov    DWORD PTR [esp+0x1c],0x0
12    0x080483f5 <+17>:    jmp    0x8048411 <main+45>
13    0x080483f7 <+19>:    mov    eax,0x80484f0
14    0x080483fc <+24>:    mov    edx,DWORD PTR [esp+0x1c]
15    0x08048400 <+28>:    mov    DWORD PTR [esp+0x4],edx
16    0x08048404 <+32>:    mov    DWORD PTR [esp],eax
17    0x08048407 <+35>:    call   0x8048300 <printf@plt>
18    0x0804840c <+40>:    add    DWORD PTR [esp+0x1c],0x1
19    0x08048411 <+45>:    cmp    DWORD PTR [esp+0x1c],0x3e7
20    0x08048419 <+53>:    jle    0x80483f7 <main+19>
21    0x0804841b <+55>:    leave
22    0x0804841c <+56>:    ret
23 End of assembler dump.
24 (gdb)
```

In line 4 gdb's disassembly syntax is set to Intel's which is of the following form:
operation <destination> <source>
This was done, since Intel's syntax is thought to be easier to be read compared to AT&T's syntax which is the default. As it can be derived, the "for loop" starts at address 0x080483f5 (+17) where a jump instruction sends execution to 0x8048411 (+45). The instruction there compares the contents of i to 0x3e7 (999_{10}) and while i is less than or equal to 999 execution will loop among the instructions placed in memory at addresses from 0x080483f7 (+19) to 0x08048419 (+53) until the condition is met (i.e. i is not less or equal to 999) and execution is completed. By calculating the difference of the two addresses that the loop is placed within ($|0x080483f7 - 0x08048419|$[9]) we obtain the size of the instructions that constitute the loop, which equals to 34 bytes. 34 bytes are enough to fit in one 4KB page. Moreover, it can be observed that the whole main function fits in a single page since the total size of its instructions ($|0x080483e4 - 0x0804841c|$) are 56 bytes. Furthermore, since main() will be called by some other function the compiler has created, it is expected that the page where all the instructions that constitute the main function are contained will have been loaded in TLB before main starts executing. Therefore, 0 or 1 misses are expected. 1 miss might occur since instructions may have been loaded at the end of one page and might continue to the next. Thus, 1 miss would occur while accessing the next page. However, this is rare.

By executing Cachegrind with itlb-friendly.c and both dTLB and L2TLB disabled, the following results are obtained:

```
1  $valgrind --tool=cachegrind  --dTLB=-1,-1,-1 --L2TLB=-1,-1,-1 ./itf
2  ...(Cachegrind output cropped)...
3  ==20589== ---Results---
4  ==20589==
5  ==20589== ---iTLB Stats---
6  ==20589== Total Accesses:   966423
7  ==20589== Hits:             966370
8  ==20589== Misses:           53
9  ==20589== Hit ratio:        99.9%
10 ==20589== Miss ratio:       0.0%
```

[9]—number— indicate the absolute value of the number.

Indeed the number of misses is very small (53). A more detailed analysis can take place using cg_annotate by limiting results to be instruction related:

```
$cg_annotate cachegrind.out.20589 --show=TIr,TI1mr,TI2mr ~/Desktop/itlb-friendly.c
...(cg_annotate output cropped)...
--------------------------------------------------------------------------------
-- User-annotated source: /home/(..)/Desktop/itlb-friendly.c
--------------------------------------------------------------------------------
  TIr TI1mr TI2mr

    .     .     .  #include <stdio.h>
    .     .     .
    4     0     0  int main(){
    .     .     .
    .     .     .       int i;
3,004     0     0       for(i=0;i<1000;i++)
5,000     0     0               printf("%d\n",i);
    .     .     .
    2     0     0  }
```

As it can be seen the number of instruction misses is 0, as expected.

iTLB Unfriendly code

We now consider an iTLB-unfriendly code. iTLB-unfriendly code is achieved by creating code whose instructions are big enough so that they do not fit in the TLB and they are not close to each other in memory. A program that implements the above ideas is one that contains a huge switch statement of hundreds of thousands of numbers. This program was created with the aid of a script. Since the resulted program is very large and cannot be shown, its structure will be shown in order to familiarise the reader with it. itlb-unfriendly.c is shown below and explanation follows:

Listing 4.4: itlb-unfriendly.c

```
1
2  #define SWITCH_NUM (200000)
3
4  void switchFun(int i){
5      int z=0;
6
7      switch(i){
8
9          case 0:
10             z+=some_random_value;
11             break;
12         case 1:
13             z+=some_random_value;
14             break;
15         case 2:
16             z+=some_random_value;
17             break;
18         ...
19         ...
20         //more cases
21         ...
22         case SWITCH_NUM-1:
23             z+=some_random_value;
24             break;
25         }
26 }
27
28 int main(){
```

```
29      int r,i;
30      for(i=0;i<SWITCH_NUM;i++){
31          r+=1024;
32          if(r>SWITCH_NUM)
33              r=0;
34          switchFun(r);
35      }
36      return 0;
37  }
```

The main function loops SWITCH_NUM times and calls the function switchFun(..) with a parameter r that is increased by 1024 every time the loop is executed. If r gets bigger than SWITCH$_NUMitissetbackto$0.

switchFun(..) contains a huge switch statement with SWITCH_NUM number of cases, where each case assigns a random value to a variable z.

Initially, instead of a switch statement hundreds of thousands of functions were created and called. However, some level of optimisation took place every time, even though optimisations were disabled, whose origin could not be identified. Thus, a switch statement was chosen instead, with each case containing an assignment to a random number so any optimisation attempt would fail due to the different values assigned in z in each case. Compiler optimisation was disabled again. As examined by gdb the total size of the instructions consisting switchFun() are 2,399,609 bytes. Recall that the page size is 4096 bytes. Each case option corresponds to instructions consisting of 7 bytes in total. Thus, each page can contain 586 (=4096/7) cases. Therefore, if case 0 is accessed, then any case access from 0 to 585 would result in a TLB hit. In order to avoid that, the variable r is increased by 1024 so that every case access will result in a miss. Moreover, if r gets big enough to correspond to a non existing case, it is set back to 0.

Since every time a non cached case is accessed, it is expected that the total number of misses would be equal to the total number of accesses which are set by SWITCH_NUM and are 200,000.

By compiling[10] and running Cachegrind upon itlb-unfriendly.c program the following statistics are obtained:

```
1  $gcc -O0 -fno-inline-functions -g -o tuf itlb-unfriendly.c
2  $valgrind --tool=cachegrind --dTLB=-1,-1,-1 --L2TLB=-1,-1,-1 ./tuf
3  ...(Cachegrind output cropped)...
4  ==20841== ---Results---
5  ==20841==
6  ==20841== ---iTLB Stats---
7  ==20841== Total Accesses:   4909908
8  ==20841== Hits:             4710879
9  ==20841== Misses:           199029
10 ==20841== Hit ratio:        95.9%
11 ==20841== Miss ratio:       4.0%
```

As it can be seen from the above results in line 9, misses (199,029) are very close to the expected number (200,000). Again, cg_annotate can be used to provide more information about the origin of the misses:

```
1  $cg_annotate cachegrind.out.20841 --show=TIr,TI1mr ~/Desktop/itlb-↩
       unfriendly.c
2  ...(cg_annotate output cropped)...
3  --------------------------------------------------------
4         TIr    TI1mr  file:function
5  --------------------------------------------------------
6  3,000,000 198,979  /home/(..)/Desktop/itlb-unfriendly.c:switchFun
7  1,801,031       0  /home/(..)/Desktop/itlb-unfriendly.c:main
8    104,058      42  ???:???
```

[10] Compiling with -O0 disables optimisations, while -fno-inline-functions makes sure that no function is treated as an inline function (the latter is merely a precaution).

This time the per function statistics are shown. As it can be observed, misses in switchFun() (198,979) are very close to those expected (200,000). The same test was made for various values of SWITCH_NUM, where the misses obtained were very close to the number set in SWITCH_NUM. Specifically, tests for SWITCH_NUM = 150,000 resulted into 148,979 misses, tests for SWITCH_NUM = 100,000 resulted into 98,979 misses and tests for SWITCH_NUM = 75,000 resulted into 72,973 misses. Finally, setting SWITCH_NUM to 200,000 and increasing the r variable by 100 instead of 1024 (i.e. for every 6 hits there is one 1 miss) drops the number of misses to 58,471 which is again very close to the expected (60,000).

4.5.3 Checking L2TLB

L2TLB is unified and thus holds both instruction and data. First of all, a check has to be made upon the number of references of L2TLB to ensure that they are equal to the number of misses of iTLB and dTLB added together, i.e. to check that every miss in L1 TLBs is passed onto L2TLB. Secondly, a check has to be made to ensure that L2TLB produces correct results.

In order to check that the number of L2TLB references equals the number of L1 TLBs' misses a program like `ls` could be tested. Execution results are shown below and analysis follows:

```
1  ...(Cachegrind output cropped)...
2  ==3260== ---Results---
3  ==3260==
4  ==3260== ---iTLB Stats---
5  ==3260== Total Accesses:   1033844
6  ==3260== Hits:             1033694
7  ==3260== Misses:           150
8  ==3260== Hit ratio:        99.9%
9  ==3260== Miss ratio:       0.0%
10 ==3260==
11 ==3260==
12 ==3260== ---dTLB Stats---
13 ==3260== Total Accesses:   519453
14 ==3260== Hits:             519065
15 ==3260== Misses:           388
16 ==3260== Hit ratio:        99.9%
17 ==3260== Miss ratio:       0.0%
18 ==3260==
19 ==3260==
20 ==3260== ---L2TLB Stats---
21 ==3260== Total Accesses:   538
22 ==3260== Hits:             313
23 ==3260== Misses:           225
24 ==3260== Hit ratio:        58.1%
25 ==3260== Miss ratio:       41.8%
```

The number of iTLB misses is 150, while the number of dTLB misses is 388. Adding these two together equals 538 which is indeed the number of L2TLB total accesses. The same test was done using many different programs and the results were exactly as expected. Thus, it is considered that the simulator correctly passes any L1 TLB miss occurrence onto L2TLB.

It should be noticed that executing `ls` (and any program in general) with L2TLB produces different number of misses compared to not having a L2TLB. For `ls` the difference can be seen in the following table:

Table 4.2: iTLB and dTLB results with and without L2TLB

	Without L2TLB	**With L2TLB**
iTLB		

Continued on next page

Table 4.2 – continued from previous page

	Without L2TLB	**With L2TLB**
Total Accesses	1053568	1033844
Hits	1053418	1033694
Misses	150	150
Hit Ratio	99.9%	99.9%
Miss Ratio	0.0%	0.0%
dTLB		
Total Accesses	530987	519453
Hits	530716	519065
Misses	271	388
Hit Ratio	99.9%	99.9%
Miss Ratio	0.0%	0.0%
L2TLB		
Total Accesses	-	538
Hits	-	313
Misses	-	225
Hit Ratio	-	58.1%
Miss Ratio	-	41.8%

It should be noticed that the number of accesses is different per execution for reasons explained earlier.

For the given example, the two executions differ in the number of dTLB misses. This is caused due to how misses are treated. An L1TLB miss is passed onto L2TLB and if found, execution will continue without performing any replacements. In case of a L2TLB miss, the tag missed will be replaced in both L1 and L2TLB. However, L2TLB is much larger in size (1024 entries compared to 64 entries of L1TLBs) so it will contain more tags. After a number of L1 and L2 TLB misses have taken place, if a L1 TLB miss occurs, since L2TLB has larger capacity than L1 TLBs, chances are that a tag will not have been overwritten and it will be found there. Therefore, that tag it will not be replaced in L1. Moreover, if the same tag is used many times, this might result to few L1 hits and to many L2 hits.

A fictitious example is used to illustrate the above point. We assume that the only existing TLBs are dTLB and L2TLB. Initially, both are empty. Moreover, we assume a TLB look up of a tag whose value is X. Since both TLBs are empty, the look up will result in a miss, thus, the tag will be entered in both TLBs. Then, we assume a TLB look up of a tag whose value is Y that happens to overwrite the X tag in dTLB, but not in L2TLB. Now, dTLB only contains the X tag, while L2TLB contains both the X and the Y tags. Now, if tag X is used again many times it will result to many dTLB misses and many L2TLB hits. That explains why the number of misses in L1 TLBs might be different when simulation is done with and without L2TLB.

Recall dtlb-unfriendly.c which, as explained earlier, causes 3,072,200 misses in dTLB from every array cell accessed. A dTLB miss in that case will result in a L2TLB miss, since for every array cell a new page is accessed which could not exist in any of the two TLBs. By neglecting iTLB results, it is expected that the number of L2TLB references will be at least[11] 3,072,200. Moreover, each of these 3,072,200 references should result in a miss. Therefore, L2TLB misses are expected to be at least 3,072,200 and the miss ratio is expected to be high (although, it is not necessary that it will be high). By checking the above assumption with Cachegrind, we obtain the following results that analysed afterwards:

```
1 $gcc -g -o dtu dtlb-unfriendly.c
2 $valgrind --tool=cachegrind --iTLB=-1,-1,-1 ./dtu
3 ==3290== ---Results---
4 ==3290==
```

[11] Saying "at least", because some other misses might occur from other parts of the program.

```
==3290== ---dTLB Stats---
==3290== Total Accesses:   15414633
==3290== Hits:             12342561
==3290== Misses:           3072072
==3290== Hit ratio:        80.0%
==3290== Miss ratio:       19.9%
==3290==
==3290==
==3290== ---L2TLB Stats---
==3290== Total Accesses:   3072072
==3290== Hits:             0
==3290== Misses:           3072072
==3290== Hit ratio:        0.0%
==3290== Miss ratio:       100.0%
```

As it can be seen, results are along the lines of the expected ones. The program caused 3,072,200 + 72 misses all of which resulted in a miss. L2TLB misses were expected to be at least 3,072,200 which is true and moreover the miss ratio is very high, up to 100

4.5.4 Conclusion on Result's Validity

For all the tests made, the corresponding results were along the lines of the expected ones. Therefore, it is concluded that TLB measuring extensions work correctly for a variety of cases and results are very close or equal to the expected ones. However, we cannot assume that the extension is totally correct since it cannot be proven that results are correct, and besides that, errors might be found at a later stage that had not be noticed so far.

4.6 Measuring Execution Time

As stated in the corresponding section of L2 cache inclusion, one of the initial requirements set before extending Cachegrind was not to introduce a major slowdown in its performance. Now that TLB measuring has been completed, some tests should be performed to observe the amount of slowdown introduced. Execution time measurement was done in two parts. First of all, normal programs, that do not have —- or they have no reason to have — an unfriendly TLB behaviour were executed first. Secondly, the TLB specific programs described in "*4.5 — Validity of Results*" section were executed. The two different tests were done in order to estimate the slowdown occurred in "normal" programs and to those that have a specific TLB behaviour.

All tests were done by timing execution times 10 times and calculating their average. Time was measured using the built-in Unix program `/usr/bin/time` ("user time"). Results collected for "normal" programs are shown in figure 4.11. The X axis represents the different programs executed, while the Y axis presents the execution time of each program. As it can been observed, simple TLB measuring times are higher than the original's Cachegrind execution times. Moreover, TLB measuring with page tracking times are higher than simple TLB measuring times.

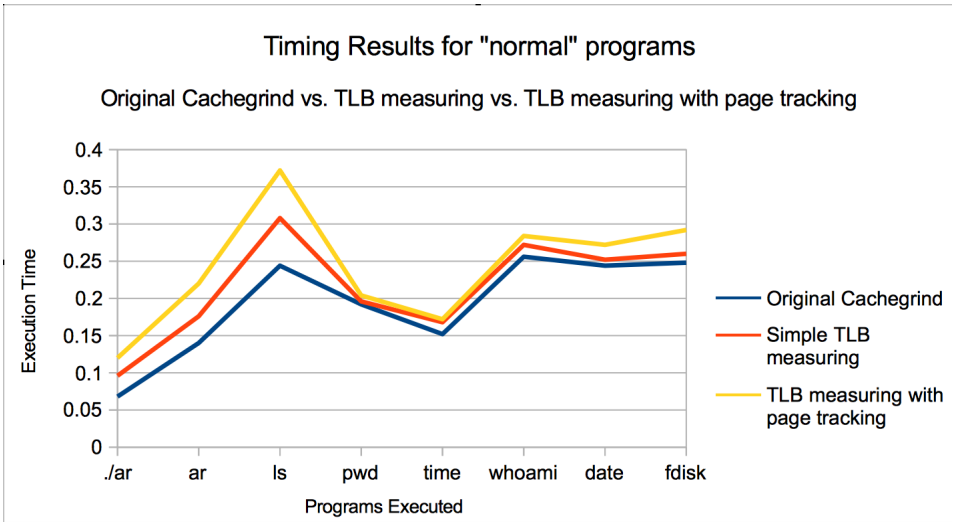

Figure 4.11: Timing results for normal programs

For simple TLB measuring, the maximum slowdown observed was 41% produced by ./ar (array.c), while the minimum slowdown was 2% produced by pwd.

For TLB measuring with page tracking enabled, the maximum slowdown introduced was 76% produced by ./ar, while the minimum was again produced by pwd (6%).

Execution times regarding TLB specific programs can be seen in figure 4.12. This time only four programs were executed. As it can be seen, dtlb-unfriendly.c when executes with page tracking introduces a huge slowdown which reached 1400% of original Cachegrind's time. On the other hand, simple TLB measuring introduced a maximum slowdown of 440%.

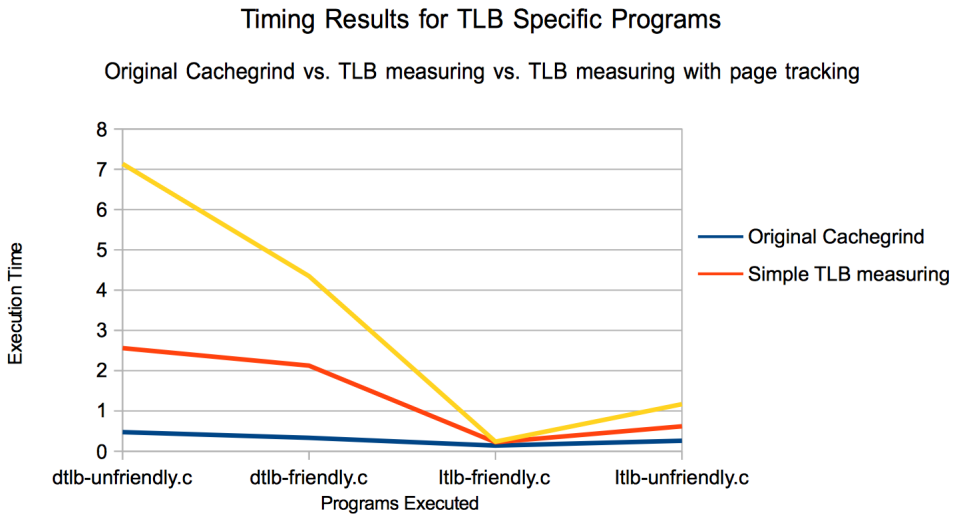

Figure 4.12: Timing results for TLB specific programs

4.6.1 Conclusion on Time

As results indicated, TLB measuring takes more time to execute. In most executions regarding "normal" programs, execution time was close to the original version, however, some executions produce a greater amount of slowdown. The maximum slowdown percentage noticed for simple TLB measuring was 41%, while for TLB measuring with page tracking was 76%. Moreover, executions regarding TLB unfriendly programs introduced a slowdown of 440% for simple TLB measuring, while for TLB measuring with page tracking the slowdown

percentage reached 1400%. Given the amount of slowdown introduced by TLB measuring with page tracking, it was decided that page tracking will be available as an option, since otherwise, it introduces an undesired amount of slowdown which is better to be avoided if possible.

Chapter 5

Conclusion

Upon embarkment on this project our knowledge regarding Cachegrind and the TLB was very limited. Nevertheless, we managed to understand TLB and the framework within which it is conceptualised, as well as to extend Cachegrind to provide information for L2 CPU cache and the TLB. It should be noted that all of the initial requirements were completed.

The final product can be helpful for programmers to better understand hits and misses, caused by their programs, related to CPU caches. Moreover, programmers have been given the TLB as a new topic to reason about. The combination of the above can help programmers develop even faster and more efficient programs. Furthermore, the contribution of this project can be helpful within the academic community. An already known study that the TLB measuring extension would contribute is [15]. The authors of [15] would have liked to measure the TLB, but they were in lack of a tool that could do it as desired. Our TLB extension is considered to cover the needs of the authors and could be used in a later study.

5.1 Difficulties Encountered

Difficulties were encountered in both L2 cache inclusion and TLB measuring. L2 cache inclusion was very similar to how the original version of Cachegrind, already, treated the other CPU caches and, especially, the last level cache. Thus, it was understood that we should be along the lines Cachegrind's current structure and do not deviate much. However, the very challenging part was to understand Cachegrind's code in order to be able to extend it. Cachegrind is not very well documented. No particularly user-friendly documentation exists, apart from the source code and its comments, that are fairly well documented. Therefore, we had to spent a large amount of time understanding the concepts beneath Valgrind's tools and, subsequently, understand Cachegrind.

On the other hand, when we proceeded on TLB measuring, the very challenging part was to understand how TLB works in depth, devise a method in order to simulate it and, develop the simulation code. The previous experience we had with L2 cache inclusion was very helpful, since we knew Cachegrind's structure and, the parts of code that had to be extended were easily located. Therefore, the TLB extension could be integrated into Cachegrind with limited effort.

5.2 Critique of the Validity Techniques Used

The techniques used to validate the results produced by both L2 cache inclusion and TLB measuring are not the best in terms that they do not prove that results are correct, they only check if results are along the lines of the expected ones. Although, the checks provide a first level checking that allows to distinguish if results are logical or not and, if not, determine errors, the lack of a mathematical proof does not allow the extensions to be called totally correct.

The reason that checks were chosen instead of a mathematical proof is because a proof was very difficult to be obtained, since a number of factors, that we may not be able to conceptualise, may affect the result and, thus, we thought that trying to prove the validity of the results would fail. Therefore, we have chosen to show that results are logical and admit that they may be prone to errors.

5.3 Lessons Learnt

The lessons learnt during engagement with the project regard several aspects. First of all, the project had to be subject to detailed schedule plan and continuous work was necessary in order to be completed. However, as it is a well known secret in the Computer Science field, it is very difficult to strictly adhere to a scheduled plan, since difficulties or errors appear. Our schedule plan was devised with that kind of unexpected events in mind and, thus, we had plenty of time to handle them when they appeared. The lesson verified in this case is that we should always plan with unexpected events in mind, so there will be enough time to handle any problems.

A second aspect learnt upon engagement with the project is knowledge of low level aspects of the computer. As stated, our TLB knowledge was limited, but was extended during the project. Moreover, by reading about Cachegrind and memory concepts we started taking seriously into consideration the effect cache friendliness has on execution speed and that creating cache friendly programs is an important skill. Furthermore, by studying Cachegrind's source code we came up with C code "tricks" regarding speed increase and security checks that we had not encountered ever before. These include, but are not limited to, an interesting use of inline functions and goto statements.

The last, but not least, aspect learnt is how to extend a project. This includes how to read the source code and alter it. This is the first time we extended a project developed by a third person. Although, doing so, it pointed out that documentation, comments, as well as the name of the variables, are crucial when developing a program. This is true as for developers themselves, as it is for the people that might extend the code. Finally, extending a project developed by someone else might often be the case in the future, therefore, our extension is considered to be an important experience.

5.4 Future Work

In order to conclude with the project, we present some aspects that can be further extended in order to make Cachegrind a better and more efficient tool. Some of them require less effort to be implemented than others, while, some are more important than others. A list of the extensions that can take place within Cachegrind is shown below:

- The extensions described in this project were Intel x86 specific. Cachegrind could be extended to work on a variety of architectures, such as AMD, powerPC and others.
- The part of Cachegrind that is related to simulating CPU caches could be changed in order to be extended easier. Its current structure it is not a product of good software engineering, since many separate variables have to be created in order to measure different aspects of the system, while, a main data structure that would contain all information necessary to simulate the CPU caches could have been used instead, as it is the case in TLB measuring.
- Nowadays, many computers have four level of CPU caches. Cachegrind provides information for three levels only, however, it could be further extended to include a fourth level of CPU cache.
- The TLB simulator can only detect configurations related to 4KB sized pages, since the page size used by the system is not accessible in user-mode. A module could be built that runs in kernel-mode and obtains the current page size. By making the appropriate extensions, Cachegrind could be able to simulate any TLB configuration.

- Cachegrind could be extended to account for parallel computing. This includes being able to simulate programs that run in more than one core, as well as to consider the fact that some CPU caches and L2TLB are shared and, thus, their contents can be changed by other programs.
- Cachegrind could be extended to account for context switches. Since Cachegrind is a simulator, it does not have to determine the context switches that occur during the simulation, but, instead, it could measure the number of context switches that may happen during a program's execution and the effect they would have on program's performance. Taking context switches into consideration will provide a more accurate picture of the simulation of both CPU caches and the TLB, however, results will not change dramatically.
- The TLB simulator could be further extended to be faster. Increase in speed could be achieved by using inline functions, by making the code more compact and efficient and by using OSet for page tracking instead of a linked list, since OSet is more efficient and faster.

Bibliography

[1] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006, pp. 454–457. ISBN: 0321486811.

[2] F. Alted. "Why Modern CPUs Are Starving and What Can Be Done About It". In: *Computing in Science and Engg.* 12.2 (Mar. 2010), pp. 68–71. ISSN: 1521-9615. DOI: 10.1109/MCSE.2010.51. URL: http://dx.doi.org/10.1109/MCSE.2010.51.

[3] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. 2013, pp. 105–209.

[4] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. 2nd. USA: Addison-Wesley Publishing Company, 2010, pp. 775–807. ISBN: 0136108040, 9780136108047.

[5] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference, A-M*. Last Accessed 3/4/2014. Intel Corporation. 2014, p. 572. URL: http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-2a-manual.pdf.

[6] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B & 3C): System Programming Guide*. Intel Corporation. 2012, pp. 121–128,393–397.

[7] Intel Corporation. *Intel® Processor Identification and the CPUID Instruction*. Accessed on 3/12/2013. Intel Corporation. 2002, pp. 7–11. URL: http://datasheets.chipdb.org/Intel/x86/CPUID/24161821.pdf.

[8] MSDN Developers. *Introduction to Instrumentation and Tracing*. Website. Accessed on 1/3/2014. 2014. URL: http://msdn.microsoft.com/en-us/library/x5952w0c(v=vs.110).aspx.

[9] Valgrind Developers. *About Valgrind*. URL: http://valgrind.org/info/about.html.

[10] Valgrind Developers. *Cachegrind: a cache and branch-prediction profiler*. URL: http://valgrind.org/docs/manual/cg-manual.html.

[11] Valgrind Developers. "Valgrind Documentation". PDF file within Valgrind 3.8.1 archive. 2012.

[12] Ulrich Drepper. *What Every Programmer Should Know About Memory*. 2007. URL: http://people.freebsd.org/~lstewart/articles/cpumemory.pdf.

[13] James M. Feldman and Charles Retter. *Computer Architecture; A Designer's Text Based on a Generic RISC*. 1st. New York, NY, USA: McGraw-Hill, Inc., 1993. ISBN: 0070204535.

[14] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. 5th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011, pp. 578–593. ISBN: 012383872X, 9780123838728.

[15] Jessica R. Jones, James H. Davenport, and Russell Bradford. "The Changing Relevance of the TLB". In: *Proceedings of the 2013 12th International Symposium on Distributed Computing and Applications to Business, Engineering & Science*. DCABES '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 110–114. ISBN: 978-0-7695-5060-2. DOI: 10.1109/DCABES.2013.27. URL: http://dx.doi.org/10.1109/DCABES.2013.27.

[16] Chi-Keung Luk et al. "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation". In: *SIGPLAN Not.* 40.6 (June 2005), pp. 190–200. ISSN: 0362-1340. DOI: 10.1145/1064978.1065034. URL: http://doi.acm.org/10.1145/1064978.1065034.

[17] David Mosberger and Stâephane Eranian. *Ia-64 Linux Kernel: Design and Implementation*. section 4.4. Prentice Hall PTR, 2002.

[18] Andreas Moshovos. *Notes on Caches*. Lecture Notes. Last Accessed 3/4/2014. 2007. URL: http://www.eecg.toronto.edu/~moshovos/ECE243-07/l26-caches.html.

[19] David R. Musser and Noboru Obata. *A Portable Cache Profiler Based on Source-Level Instrumentation*. 2003, pp. 1–8.

[20] Nicholas Nethercote. Cachegrind's Source Code Comments Version 3.8.1.

[21] Nicholas Nethercote. *Dynamic binary analysis and instrumentation*. Tech. rep. UCAM-CL-TR-606. University of Cambridge, Computer Laboratory, Nov. 2004, pp. 11–77. URL: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-606.pdf.

[22] Nicholas Nethercote and Julian Seward. "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation". In: *SIGPLAN Not.* 42.6 (June 2007), pp. 89–100. ISSN: 0362-1340. DOI: 10.1145/1273442.1250746. URL: http://doi.acm.org/10.1145/1273442.1250746.

[23] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*. 4th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008, pp. 452–518. ISBN: 0123744938, 9780123744937.

Appendix A

TLB Measuring Source Code

This appendix only contains the source code related to TLB measuring. The reason that not all Cachegrind's source code is included is because it would occupy many pages. Therefore, the most interesting part of the code was chosen to be presented here.

A.1 cg_tlb.c

```
1
2  /*↩
       ------------------------------------------------------------------↩
       */
3  /*--- TLB Measuring & Page Tracking Tool                    cg_tlb.c ↩
       ---*/
4  /*↩
       ------------------------------------------------------------------↩
       */
5
6  /*
7   This file is part of Cachegrind, a Valgrind tool for cache
8   profiling programs.
9
10  Copyright (C) 2002-2012 Nicholas Nethercote
11  njn@valgrind.org
12
13  This program is free software; you can redistribute it and/or
14  modify it under the terms of the GNU General Public License as
15  published by the Free Software Foundation; either version 2 of the
16  License, or (at your option) any later version.
17
18  This program is distributed in the hope that it will be useful, but
19  WITHOUT ANY WARRANTY; without even the implied warranty of
20  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
21  General Public License for more details.
22
23  You should have received a copy of the GNU General Public License
24  along with this program; if not, write to the Free Software
25  Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA
26  02111-1307, USA.
27
28  The GNU General Public License is contained in the file COPYING.
29  */
30
31
32 #include "pub_tool_basics.h"
```

```
33 #include "pub_tool_tooliface.h"
34 #include "pub_tool_libcassert.h"
35 #include "pub_tool_libcprint.h"
36 #include "pub_tool_debuginfo.h"
37 #include "pub_tool_libcbase.h"
38 #include "pub_tool_options.h"
39 #include "pub_tool_machine.h"     // VG_(fnptr_to_fnentry)
40 #include "pub_tool_mallocfree.h"
41
42
43 #define TLB_TYPE_ITLB  (0)
44 #define TLB_TYPE_DTLB  (1)
45 #define TLB_TYPE_L2TLB (2)
46
47
48 /*------------------------------------------------------------*/
49 /*--- Paging Information                                   ---*/
50 /*------------------------------------------------------------*/
51
52 /* This structure is used to hold all pages accessed and the times of↩
       access.
53  * This is done only to show to the user.
54  * To be used only if option --show-pages is enabled.
55  * It is set dynamically (linked list), since the number of pages ↩
       cannot be known from before */
56 typedef struct pages_accessed{
57     Addr tag;
58     ULong count; //ULong cause it might get big enough
59     struct pages_accessed *next;
60 }PAGES;
61
62
63 /*------------------------------------------------------------*/
64 /*--- TLB Simulation                                       ---*/
65 /*------------------------------------------------------------*/
66
67 /* Holds TLB characteristics */
68 typedef struct _tlb_t{
69
70     //Holds hits and misses per TLB
71     Int hit,miss;
72
73     //Page size should always be in B.
74     //i.e. if page size is 4KB then PAGE_SIZE=4*1024=4096
75     ULong page_size;
76
77     /* assoc represents associativity.
78      * -1 is Fully Associative
79      *  0 is Direct Mapped
80      *  N where N>0 is N-way Associative */
81     Int assoc;
82     Int entries;
83
84     ULong    offset_mask;    //Used to separate Virtual Address ↩
           Offset
85     ULong    vpn_mask;       //Used to separate Virtual Address VPN
86     ULong    index_mask;     //used in direct map, to split VPN to ↩
           tag and index.
87     ULong    tag_mask;       //Used in direct map, to get the tag ↩
           from VPN.
```

```
88      Int       sets;           //Used in N-way assoc, to hold the ↩
            number of sets
89
90      Bool      replace;        //True if TLB was full and from now on a↩
             replacing algorithm has to choose what to replace. Used in ↩
            FA.
91
92      PAGES     *page_ptr;
93      Int       total_pages;
94
95      Int       tlb_counter;    //Points to which entry is currently the↩
             TLB operating
96
97      Char     desc_line[128];  //Holds the tlb configuration in words.
98  }tlb_t;
99
100
101 /*
102  *TLBc[0] -> iTLB
103  *TLBc[1] -> dTLB
104  *TLBc[2] -> L2TLB
105  */
106 static tlb_t TLBc[3];
107
108
109 /* This structure represents a TLB entry.
110  * Holds the tag and the times it was accessed.
111  * An array of TLB_ENTRYs consists a TLB */
112 typedef struct _tlb_entry{
113     Addr tag;
114     Int count;
115 }TLB_ENTRY;
116
117 //TLB Array
118 TLB_ENTRY **TLB;
119
120
121 /*-------------------------------------------------------------*/
122 /*--- Command Line Arguments                                ---*/
123 /*-------------------------------------------------------------*/
124 static Bool clo_sim_tlb=True;       /* Simulate TLB     */
125 static Bool clo_sim_pages=False;    /* Simulate Pages   */
126
127 /*-------------------------------------------------------------*/
128 /*--- Simulation General Info                               ---*/
129 /*-------------------------------------------------------------*/
130
131 /* Virtual Address Space Size */
132 static Int VAS_SIZE=32; //initially set to 32 since it's the most ↩
        common.
133
134 /* Replacement Policy */
135 /*
136  * 0 -> LFU
137  * 1 -> LRU
138  * 2 -> RR
139  */
140 static Int RepPol=1; //LRU is the default
141
142 /* Which TLBs to simulate? (iTLB, dTLB or L2TLB) */
143 static Bool sim_iTLB=False;
```

```
144 static Bool sim_dTLB=False;
145 static Bool sim_L2TLB=False;
146
147 /* TLB_TYPE distinguishes between iTLB, dTLB, L2TLB (unified) or all.
148  * -1 -> all   //Not used anymore
149  *  0 -> iTLB
150  *  1 -> dTLB
151  *  2 -> L2TLB */
152 static Int TLB_TYPE=-1; //this will change soon enough to point to ↩
        the valid TLB. No worries it's -1.
153
154
155 /*-------------------------------------------------------------*/
156 /*--- Function Definitions                                  ---*/
157 /*-------------------------------------------------------------*/
158
159 Int log2(Int);
160 void add_page(Addr);
161 void print_pages(void);
162
163 void do_miss(ULong*, ULong*);
164 void do_hit(void);
165
166 void TLB_simulation(Addr, ULong*, ULong*);
167 void tlb_lookup(Addr, Addr, Addr, ULong*, ULong*);
168
169 void reference_address(Addr, Int, ULong*, ULong*);
170
171 Int LFU(Int);
172 Int LRU(Int);
173 void increase_LRU(Int);
174 Int get_random(Int);
175
176 void print_tlb(Int);
177 void print_tlb_contents(void);
178 void tlb_chars(void);
179 ULong calc_VPN_MASK(Int, ULong);
180 void tlb_post_clo_init(void);
181 void print_stats(Int, Int);
182 void tlbsim_init(Int, Int, Int, Int);
183 Bool isTLBsim(void);
184
185 /*↩
        ----------------------------------------------------------------------------------------
        */
186
187 Bool isTLBsim(void){
188     return clo_sim_tlb;
189 }
190
191 /*↩
        ----------------------------------------------------------------------------------------
        */
192
193 /* Adds a page in PAGES linked list */
194 void add_page(Addr tag){
195
196     PAGES *cur=TLBc[TLB_TYPE].page_ptr;
197
198     //search if Page already exists
199     while(cur!=NULL){
```

```
200         if(cur->tag==tag){
201             cur->count++;
202             return;
203         }
204         cur=cur->next;
205     }
206
207     // Create new page.
208     cur=(PAGES*)VG_(malloc)("pages.1",sizeof(PAGES));
209     cur->next=TLBc[TLB_TYPE].page_ptr;
210     cur->tag=tag;
211     cur->count=1;
212     TLBc[TLB_TYPE].page_ptr=cur;
213     TLBc[TLB_TYPE].total_pages++;
214     return;
215 }
216
217 /*↩
        ----------------------------------------------------------------------------------
        */
218
219 /* Called by cg_fini() to print page information on screen */
220 /* And to free all page entries */
221 void print_pages(void){
222     VG_(umsg)("---Pages Accessed---\n");
223     Int i;
224     for(i=0;i<3;i++){
225         if( (sim_iTLB && i==0) || (sim_dTLB && i==1) || (sim_L2TLB &&↩
                i==2)){
226
227             switch(i){
228                 case 0: VG_(umsg)("\niTLB Pages Accesed\n"); break;
229                 case 1: VG_(umsg)("\ndTLB Pages Accesed\n"); break;
230                 case 2: VG_(umsg)("\nL2TLB Pages Accesed\n"); break;
231             }
232             //Print total pages accessed
233             VG_(umsg)("Pages Accessed In total:   %d\n",TLBc[i].↩
                    total_pages);
234
235             //print each page and times it was accessed
236             Int i2=0;
237             PAGES *cur=TLBc[i].page_ptr;
238             while(cur!=NULL){
239                 VG_(umsg)("%d) Page %08lx, accessed %llu times\n",i2↩
                        +1,cur->tag,cur->count);
240                 i2++;
241                 cur=cur->next;
242             }
243
244             //free all malloced pages
245             while(TLBc[i].page_ptr!=NULL){
246                 cur=TLBc[i].page_ptr->next;
247                 VG_(free)(TLBc[i].page_ptr);
248                 TLBc[i].page_ptr=cur;
249             }
250             //VG_(umsg)("Freed everything\n");
251         }
252     }
253
254 }
255
```

```
256 /*↩
        ------------------------------------------------------------------------------
        */
257
258 void print_tlb_contents(void){
259
260     Int i;
261     for(i=0;i<TLBc[TLB_TYPE].entries;i++)
262         VG_(umsg)("%d.|_____%d_____|\n",i,TLB[TLB_TYPE][i].tag);
263 }
264 /*↩
        ------------------------------------------------------------------------------
        */
265
266 //set target count equal to 0
267 //increase everything else by 1
268 void increase_LRU(Int target){
269
270
271     Int i;
272
273     //Fully associative
274     /* For FA increase all entries by 1 and set most recently used to↩
            0 */
275     if(TLBc[TLB_TYPE].assoc==-1){
276
277         //Increase count in all elements by one
278         for(i=0;i<TLBc[TLB_TYPE].entries;i++){
279             TLB[TLB_TYPE][i].count++;
280         }
281
282         //set target count to 0, as it is the most recently used
283         TLB[TLB_TYPE][target].count=0;
284
285     }
286
287     //N-way set associative
288     /* For Nway increase count in all entries in the set by 1 and set↩
            the most recently used count to 0*/
289     if(TLBc[TLB_TYPE].assoc>0){
290
291         //find the set in which the target belongs
292         Int set=(Int)(target/TLBc[TLB_TYPE].assoc);
293
294         //increase count in elements in the set
295         for(i=set*TLBc[TLB_TYPE].assoc;i<(set+1)*TLBc[TLB_TYPE].assoc↩
                ;i++)
296             TLB[TLB_TYPE][i].count++;
297
298         //set target count to 0
299         TLB[TLB_TYPE][target].count=0;
300     }
301
302 }
303
304 /*↩
        ------------------------------------------------------------------------------
        */
305
306 /* LRU - Least Recently Used
```

```
307   Returns the position of the TLB element that was used the least ↩
          recently */
308  Int LRU(Int set){
309
310      Int pos=-1,i,max; //max holds the max count i.e. the Least ↩
             Recently Used
311
312      /* For fully associative, scan all and return the biggest */
313      if(TLBc[TLB_TYPE].assoc==-1){
314          //set max to equal the first
315          max=TLB[TLB_TYPE][0].count;
316          //making sure pos is equal 0. This is only a pre caution for ↩
                 later changes, since otherwise,
317          //it might go unnoticed
318          pos=0;
319
320          for(i=1;i<TLBc[TLB_TYPE].entries;i++){
321              if(TLB[TLB_TYPE][i].count>max){
322                  max=TLB[TLB_TYPE][i].count;
323                  pos=i;
324              }
325          }
326          return pos;
327      }
328
329
330      if(TLBc[TLB_TYPE].assoc>0){
331          //set max to equal the count in first item in the set
332          max=TLB[TLB_TYPE][set*TLBc[TLB_TYPE].assoc].count;
333
334          //set pos to equal the first item
335          pos=set*TLBc[TLB_TYPE].assoc;
336
337
338          for(i=set*TLBc[TLB_TYPE].assoc+1;i<TLBc[TLB_TYPE].assoc*(set↩
                 +1);i++){
339              if(TLB[TLB_TYPE][i].count > max){
340                  max=TLB[TLB_TYPE][i].count;
341                  pos=i;
342              }
343          }
344          return pos;
345      }
346      return pos;
347  }
348
349
350  /*↩
         ----------------------------------------------------------------------------------
         */
351
352  /* LFU - Least Frequently Used
353   * Returns the position of the TLB element that is used the less */
354  Int LFU(Int set){
355
356      Int i,pos=-1,hold;
357
358
359      // Fully Associative
360      /* Scan all entries and return the one with the smallest count */
361      if(TLBc[TLB_TYPE].assoc==-1){
```

```
362         //hold holds the first entry count
363         hold=TLB[TLB_TYPE][0].count;
364         //making sure pos is equal 0. This is only a pre caution for ↩
                later changes, since otherwise,
365         //it might go unnoticed
366         pos=0;
367         //scan the rest and find the one with the smallest count.
368         for(i=1;i<TLBc[TLB_TYPE].entries;i++){
369             if(TLB[TLB_TYPE][i].count < hold){
370                 hold=TLB[TLB_TYPE][i].count;
371                 pos=i;
372             }
373         }
374     }
375
376
377     //N-way associative
378     /* Scan all entries in the set and return the one with the ↩
            smallest count */
379     if(TLBc[TLB_TYPE].assoc>0){
380
381         //hold should hold the count of the first item in the set
382         hold=TLB[TLB_TYPE][set*TLBc[TLB_TYPE].assoc].count;
383         //set pos to equal the first item in the set
384         pos=set*TLBc[TLB_TYPE].assoc;
385         //Scan for smaller
386         for(i=set*TLBc[TLB_TYPE].assoc+1;i<TLBc[TLB_TYPE].assoc*(set↩
                +1);i++){
387             if(TLB[TLB_TYPE][i].count < hold){
388                 hold=TLB[TLB_TYPE][i].count;
389                 pos=i;
390             }
391         }
392     }
393
394     return pos;
395 }
396
397 /*↩
        ----------------------------------------------------------------------------------------
        */
398
399 //generates a random number in the range 0-range
400 Int get_random(Int range){
401     return VG_(random)(NULL)%range;
402 }
403
404 /*↩
        ----------------------------------------------------------------------------------------
        */
405
406 void do_hit(void){
407     TLBc[TLB_TYPE].hit++;
408 }
409
410 /*↩
        ----------------------------------------------------------------------------------------
        */
411
412 void do_miss(ULong *t1, ULong *t2){
413
```

```
414     TLBc[TLB_TYPE].miss++;
415     if(TLB_TYPE==TLB_TYPE_ITLB||TLB_TYPE==TLB_TYPE_DTLB){
416         (*t1)++;
417     }
418     if(TLB_TYPE==TLB_TYPE_L2TLB){
419         (*t2)++;
420     }
421
422 }
423
424 /*↩
        ----------------------------------------------------------------------------------
        */
425
426 //checking for L2TLB?
427 Bool l2check=False;
428 //Did L2 result in a hit?
429 Bool l2hit=False;
430
431 //addr_tag should always be tag (in DM) or VPN in FA
432 //addr index
433 void tlb_lookup(Addr addr_l2,Addr addr_tag, Addr addr_index, ULong *↩
        t1, ULong *t2){
434
435
436     //fully associative. entries can go anywhere, the entire TLB is ↩
            looked up.
437     if(TLBc[TLB_TYPE].assoc==-1){
438
439         Int i=0;
440
441         //Iterate through all entries and if found increase count
442         //otherwise, deal with the miss
443         for(;i<TLBc[TLB_TYPE].entries;i++){
444             if(TLB[TLB_TYPE][i].tag==addr_tag){ //hit
445                 if(l2check)
446                     l2hit=True;
447                 do_hit();
448                 if(RepPol==0)//LFU
449                     TLB[TLB_TYPE][i].count++;
450                 if(RepPol==1)//LRU
451                     increase_LRU(i);
452                 return;
453             }
454         }
455
456         //miss
457         do_miss(t1,t2);
458
459         //if checking for l2
460         //set it l2hit=false since
461         //if execution reached this line
462         //then a miss has occurred
463         if(l2check)
464             l2hit=False;
465
466         if(sim_L2TLB){
467             Int save_tlb_type=TLB_TYPE;
468             TLB_TYPE=TLB_TYPE_L2TLB;
469             //we are now checking for L2 TLB.
470             l2check=True;
```

```
471             //avoid infinite loops
472             sim_L2TLB=False;
473
474             //tlb_lookup(addr_l2, addr_tag, addr_index, t1, t2);
475             //call tlb_simulation again for L2, to possibly use ↩
                    different masks
476             //and to save page if page tracking is on
477             TLB_simulation(addr_l2,t1,t2);
478
479             sim_L2TLB=True;
480             //We no more check for L2 TLB
481             l2check=False;
482             TLB_TYPE=save_tlb_type;
483         }
484
485         //Do replacements only if L2 TLB resulted in a miss or it ↩
                doesn't exist
486         if(sim_L2TLB==False || l2hit==False){
487             if(RepPol==0){ //LFU
488                 Int entry=LFU(-1);
489                 TLB[TLB_TYPE][entry].tag=addr_tag;
490                 TLB[TLB_TYPE][entry].count=1;
491
492             }
493             if(RepPol==1){ //LRU
494                 Int entry=LRU(-1);
495                 TLB[TLB_TYPE][entry].tag=addr_tag;
496                 TLB[TLB_TYPE][entry].count=0;
497                 increase_LRU(entry);
498
499             }
500             if(RepPol==2){ //Random
501                 //generate a random number in the range 0-entries and↩
                        use it to replace
502                 Int entry=get_random(TLBc[TLB_TYPE].entries);
503                 TLB[TLB_TYPE][entry].tag=addr_tag;
504                 TLB[TLB_TYPE][entry].count=1;
505             }
506         }
507         return;
508     }
509
510     //Direct Mapped. entries can only go (tag mod TLB_ENTRIES)
511     if(TLBc[TLB_TYPE].assoc==0){
512
513         Int entr=TLBc[TLB_TYPE].entries;
514
515         if(TLB[TLB_TYPE][addr_index%entr].tag==addr_tag){
516             if(l2check)
517                 l2hit=True;
518             do_hit();
519         }else{
520             do_miss(t1,t2);
521
522             if(l2check)
523                 l2hit=False;
524
525             if(sim_L2TLB){
526                 Int save_tlb_type=TLB_TYPE;
527                 TLB_TYPE=TLB_TYPE_L2TLB;
528                 //we are now checking for L2 TLB.
```

```
529                 l2check=True;
530                 //avoid infinite loops
531                 sim_L2TLB=False;
532 
533                 //tlb_lookup(addr_l2, addr_tag, addr_index, t1, t2);
534                 //call simulation again for L2, to possibly use ↩
                        different masks
535                 //and to save page if page tracking is on
536                 TLB_simulation(addr_l2,t1,t2);
537 
538                 sim_L2TLB=True;
539                 //We no more check for L2 TLB
540                 l2check=False;
541                 TLB_TYPE=save_tlb_type;
542             }
543 
544             //Do replacements only if L2 TLB resulted in a miss or it↩
                     doesn't exist
545             if(sim_L2TLB==False || l2hit==False){
546                 TLB[TLB_TYPE][addr_index%entr].tag=addr_tag;
547             }
548         }
549         return;
550     }
551 
552 
553     //Nway ASSOC.
554     if(TLBc[TLB_TYPE].assoc>0){
555         //Go through all entries in the set
556         Int i;
557         //if(checkL2)
558         //  VG_(umsg)("Range to check:%d-%d\n",addr_index*TLBc[↩
                TLB_TYPE].assoc,addr_index*TLBc[TLB_TYPE].assoc+TLBc[↩
                TLB_TYPE].assoc);
559 
560         for(i=0;i<TLBc[TLB_TYPE].assoc;i++){
561 
562             if(TLB[TLB_TYPE][(addr_index*TLBc[TLB_TYPE].assoc)+i].tag↩
                    ==addr_tag){ //hit
563                 if(l2check)
564                     l2hit=True;
565                 do_hit();
566                 if(RepPol==0)
567                     TLB[TLB_TYPE][(addr_index*TLBc[TLB_TYPE].assoc)+i↩
                            ].count++;
568                 if(RepPol==1)
569                     increase_LRU((addr_index*TLBc[TLB_TYPE].assoc)+i)↩
                            ;
570                 return;
571             }
572 
573         }//end for
574 
575         //Miss
576         do_miss(t1,t2);
577 
578         if(l2check)
579             l2hit=False;
580 
581         if(sim_L2TLB){
582             Int save_tlb_type=TLB_TYPE;
```

```
583             TLB_TYPE=TLB_TYPE_L2TLB;
584             //we are now checking for L2 TLB.
585             l2check=True;
586             //avoid infinite loops
587             sim_L2TLB=False;
588
589             //tlb_lookup(addr_l2, addr_tag, addr_index, t1, t2);
590             //call simulation again for L2, to possibly use different↩
                    masks
591             //and to save page if page tracking is on
592             TLB_simulation(addr_l2,t1,t2);
593
594             sim_L2TLB=True;
595             //We no more check for L2 TLB
596             l2check=False;
597             TLB_TYPE=save_tlb_type;
598         }
599
600         if(sim_L2TLB==False || l2hit==False){
601             if(RepPol==0){//LFU
602                 Int entry=LFU((Int)addr_index);
603                 TLB[TLB_TYPE][entry].tag=addr_tag;
604                 TLB[TLB_TYPE][entry].count=1;
605             }
606             if(RepPol==1){//LRU
607                 Int entry=LRU((Int)addr_index);
608                 // if(checkL2)
609                 //   VG_(umsg)("Entry:%d\n",entry);
610                 TLB[TLB_TYPE][entry].tag=addr_tag;
611                 TLB[TLB_TYPE][entry].count=0;
612                 increase_LRU(entry);
613             }
614             if(RepPol==2){//Random
615                 //generate a random number in the range 0-Nway assoc
616                 //and add it to the start of the set
617
618                 //calc start of the set
619                 Int base=addr_index*TLBc[TLB_TYPE].assoc;
620                 //calc random number in the rane 0-Nway
621                 Int entry=get_random(TLBc[TLB_TYPE].assoc);
622
623                 TLB[TLB_TYPE][base+entry].tag=addr_tag;
624                 TLB[TLB_TYPE][base+entry].count=0;
625             }
626         }
627
628         return;
629
630     }
631
632 }
633
634 /*↩
        ----------------------------------------------------------------------------------------
        */
635
636 void TLB_simulation(Addr addr, ULong *t1, ULong *t2){
637
638
639     /*
640       Virtual Address Given:
```

```
641       -------------------------
642       |       VPN       | OFFSET |
643       -------------------------
644
645       VPN:
646       -------------------------
647       |     tag       | index    |
648       -------------------------
649
650       */
651
652      //set these two to false before simulation
653      l2hit=False;
654      l2check=False;
655
656      //Extract VPN from the virtual address
657      Addr VPN=(addr & TLBc[TLB_TYPE].vpn_mask) >> TLBc[TLB_TYPE].↩
             offset_mask;
658      //alternatively:
659      //Addr VPN=addr/TLBc[TLB_TYPE].page_size;
660
661      //1) Get Hits/Misses
662      if(clo_sim_tlb||clo_sim_pages){
663
664          //Direct mapped
665          if(TLBc[TLB_TYPE].assoc==0){
666
667              /* get index */
668              // index contains only index.
669              Addr index= VPN & TLBc[TLB_TYPE].index_mask;
670
671              /* get tag */
672              //VPN now contains tag and the index bits zeroed.
673              VPN = VPN & TLBc[TLB_TYPE].tag_mask;
674              //shift tag to get the real tag number without the index ↩
                     bits
675              VPN = VPN >> log2(TLBc[TLB_TYPE].entries);
676
677              if(clo_sim_tlb)
678                  tlb_lookup(addr,VPN,index,t1,t2);
679          }//end dm if
680
681
682          //Nway Associative
683          if(TLBc[TLB_TYPE].assoc>0){
684
685              /* Get index */
686              //Only contains the index bits (i.e. the set number)
687              Addr index = VPN & TLBc[TLB_TYPE].index_mask;
688
689              /* get tag */
690              //VPN now contains tag and index bits zeroed
691              VPN = VPN & TLBc[TLB_TYPE].tag_mask;
692              //shift tag to get the real tag number without the index ↩
                     bits
693              VPN= VPN>>log2(TLBc[TLB_TYPE].index_mask+1);
694
695              if(clo_sim_tlb)
696                  tlb_lookup(addr,VPN,index,t1,t2);
697          }//end nway if
698
```

```
699
700         //Fully Associative
701         if(TLBc[TLB_TYPE].assoc==-1){
702             //In fully associative, tag=VPN, thus pass it directly.
703             if(clo_sim_tlb){
704                 tlb_lookup(addr,VPN,0,t1,t2);
705                 //tlb_lookupl2(VPN,t1,t2);
706             }
707
708         }//end fully assoc if
709
710
711     }//end if
712
713     /* 2) Save Pages */
714     //VPN now contains the tag of the address
715     if(clo_sim_pages)
716         add_page(VPN);
717
718
719
720
721 }
722
723 /*↩
        ----------------------------------------------------------------------------------------
        */
724
725
726 /* Int data_type takes the following values
727  * 0 -> iTLB
728  * 1 -> dTLB
729  */
730
731 /* Intermediary Function */
732 void reference_address(Addr addr, Int data_type, ULong *t1, ULong *t2↩
        ){
733
734     if(data_type==TLB_TYPE_ITLB && sim_iTLB){
735         TLB_TYPE=TLB_TYPE_ITLB;
736         TLB_simulation(addr,t1,t2);
737     }
738
739     if(data_type==TLB_TYPE_DTLB && sim_dTLB){
740         TLB_TYPE=TLB_TYPE_DTLB;
741         TLB_simulation(addr,t1,t2);
742     }
743 }
744
745 /*↩
        ----------------------------------------------------------------------------------------
        */
746
747 void print_tlb(Int tlb){
748
749     char *buf="";
750
751     switch(tlb){
752         case  0: buf="iTLB  (L1 Instruction TLB)";  break;
753         case  1: buf="dTLB  (L1 Data TLB)";         break;
754         default: buf="L2TLB (L2 Unified TLB)";      break;
```

```
755     }
756
757     VG_(umsg)("TLB type:          %s\n",buf);
758
759     switch(TLBc[tlb].assoc){
760         case -1: buf="Fully Associative";   break;
761         case  0: buf="Direct Mapped";       break;
762         default: buf="-Way Associative";    break;
763     }
764
765     if(TLBc[tlb].assoc>0)
766         VG_(umsg)("Associativity:     %d%s\n",TLBc[tlb].assoc,buf);
767     else
768         VG_(umsg)("Associativity:     %s\n", buf);
769
770     VG_(umsg)("Page Size:         %llu bytes\n", TLBc[tlb].page_size)↩
        ;
771     VG_(umsg)("Entries:           %d\n", TLBc[tlb].entries);
772
773
774
775 }
776
777 /*↩
    ----------------------------------------------------------------------------------------
    */
778
779 void tlb_chars(void){
780
781     VG_(umsg)("\n\n\n---TLB characteristics---\n");
782     VG_(umsg)("Virtual Address Size:     %d bits\n", VAS_SIZE);
783
784     switch(RepPol){
785         case 0: VG_(umsg)("Replacement Policy:       Least Frequently↩
                Used\n\n"); break;
786         case 1: VG_(umsg)("Replacement Policy:       Least Recently ↩
               Used\n\n"); break;
787         case 2: VG_(umsg)("Replacement Policy:       Random\n\n"); ↩
               break;
788     }
789
790     if(sim_iTLB){
791         print_tlb(0);
792         VG_(umsg)("\n\n");
793     }
794     if(sim_dTLB){
795         print_tlb(1);
796         VG_(umsg)("\n\n");
797     }
798     if(sim_L2TLB){
799         print_tlb(2);
800         VG_(umsg)("\n\n");
801     }
802
803     VG_(umsg)("\n---Results---\n\n");
804 }
805
806 /*↩
    ----------------------------------------------------------------------------------------
    */
807
```

```
808 /* This function assumes that 'num' is in power of 2*/
809 /* Returns the power to which the number should be raised */
810 Int log2(Int num){
811
812     Int log=0;
813     while(num>1){
814         num=num/2;
815         log++;
816     }
817     //VG_(umsg)("log=%d",log);
818     return log;
819 }
820
821
822 /*↩
        ----------------------------------------------------------------------------------------
        */
823
824 ULong calc_VPN_MASK(Int offset, ULong page_size){
825     Int i;
826     ULong vpn_mask=1;
827     //VG_(umsg)("offset=%d\n",offset);
828     for(i=VAS_SIZE-offset;i>0;i--){
829         vpn_mask=vpn_mask*2+1;
830     }
831
832     vpn_mask=vpn_mask*page_size;
833
834     //VG_(umsg)("VPN_MASK=%lu\n",VPN_MASK);
835     //VPN_MASK=total;
836     return vpn_mask;
837 }
838
839 /*↩
        ----------------------------------------------------------------------------------------
        */
840
841 static void init_tlb(Int tlb){
842
843     //some easy initialisations first..
844     TLBc[tlb].hit=0;
845     TLBc[tlb].miss=0;
846     //TLBc[tlb].page_size=4096;
847     TLBc[tlb].replace=False;
848     TLBc[tlb].page_ptr=NULL;
849     TLBc[tlb].total_pages=0;
850     TLBc[tlb].tlb_counter=0;
851
852     switch(TLBc[tlb].assoc){
853         case -1:
854             VG_(sprintf)(TLBc[tlb].desc_line, "%llu B, %d E, Fully ↩
                    Associative",TLBc[tlb].page_size, TLBc[tlb].entries);
855             break;
856         case 0:
857             VG_(sprintf)(TLBc[tlb].desc_line, "%llu B, %d E, Direct ↩
                    Mapped",TLBc[tlb].page_size, TLBc[tlb].entries);
858             break;
859         default:
860             VG_(sprintf)(TLBc[tlb].desc_line, "%llu B, %d E, %d-↩
                    wayAssociative",TLBc[tlb].page_size, TLBc[tlb].↩
                    entries,TLBc[tlb].assoc);
```

```
861             break;
862     }
863
864     //some more complicated now..
865
866     //calculate offset and vpn masks
867     TLBc[tlb].offset_mask=log2(TLBc[tlb].page_size);
868     TLBc[tlb].vpn_mask=calc_VPN_MASK(TLBc[tlb].offset_mask,TLBc[tlb].↩
            page_size);
869
870     Int i;
871     switch(TLBc[tlb].assoc){
872
873             //Fully Associative
874         case -1:
875             //In FA we don't need index mask nor tag_mask
876             TLBc[tlb].index_mask=-1;
877             TLBc[tlb].tag_mask=-1;
878             break;
879
880
881             //Direct Mapped
882         case 0:
883             //calculate index
884             TLBc[tlb].index_mask=1;
885             for(i=log2(TLBc[tlb].entries)-1;i>0;i--)
886                 TLBc[tlb].index_mask=TLBc[tlb].index_mask*2+1;
887
888             //tag_mask is equal to 2^(VAS_SIZE-OFFSET_BITS-1)
889             //e.g. 2^(32-12-1)=2^20 (not 2^19 because we shift bits!)
890             TLBc[tlb].tag_mask=2<<(VAS_SIZE-TLBc[tlb].offset_mask-1);
891
892             //Subtract index_mask from tag_mask
893             TLBc[tlb].tag_mask=TLBc[tlb].tag_mask-TLBc[tlb].↩
                    index_mask;
894             break;
895
896
897             //Nway Assoc
898         default:
899             //Nway can't be odd number
900             tl_assert(TLBc[tlb].assoc%2==0);
901             TLBc[tlb].sets=TLBc[tlb].entries/TLBc[tlb].assoc;
902             //VG_(umsg)("Number of sets:%d\n",SETS);
903
904             //index_mask acts as a SET MASK in this case
905             TLBc[tlb].index_mask=1;
906             for(i=log2(TLBc[tlb].sets)-1;i>0;i--)
907                 TLBc[tlb].index_mask=TLBc[tlb].index_mask*2+1;
908
909             //VG_(umsg)("INDEX_MASK=%d",INDEX_MASK);
910             TLBc[tlb].tag_mask=2<<(VAS_SIZE-TLBc[tlb].offset_mask-1);
911             TLBc[tlb].tag_mask=TLBc[tlb].tag_mask-TLBc[tlb].↩
                    index_mask;
912             break;
913
914     }
915
916 }
917
918
```

```
919 /*↩
        ------------------------------------------------------------------------------
        */
920
921 void tlb_post_clo_init(void){
922
923     TLB=(TLB_ENTRY **)VG_(malloc)("tlbg",3*sizeof(TLB_ENTRY*));
924
925     //!!!IMPORTANT
926     //init all TLBs so that we don't have to subtract to index e.g. ↩
            if itlb is not initialised
927     //then dtlb has to refer to [0] instead of [1] which messes up ↩
            everything.
928     TLB[0]=(TLB_ENTRY *)VG_(malloc)("itlb",0*sizeof(TLB_ENTRY));
929     TLB[1]=(TLB_ENTRY *)VG_(malloc)("dtlb",0*sizeof(TLB_ENTRY));
930     TLB[2]=(TLB_ENTRY *)VG_(malloc)("l2tlb",0*sizeof(TLB_ENTRY));
931
932     Int i;
933     //Initialise TLBs and allocate TLB arrays
934     if(sim_iTLB){
935         init_tlb(0);
936         TLB[0]=(TLB_ENTRY *)VG_(malloc)("itlb",TLBc[0].entries*sizeof↩
                (TLB_ENTRY));
937         //set count variable to 0
938         for(i=0;i<TLBc[0].entries;i++)
939             TLB[0][i].count=0;
940     }
941     if(sim_dTLB){
942         init_tlb(1);
943         TLB[1]=(TLB_ENTRY *)VG_(malloc)("dtlb",TLBc[1].entries*sizeof↩
                (TLB_ENTRY));
944         //set count variable to 0
945         for(i=0;i<TLBc[1].entries;i++)
946             TLB[1][i].count=0;
947     }
948     if(sim_L2TLB){
949         init_tlb(2);
950         TLB[2]=(TLB_ENTRY *)VG_(malloc)("l2tlb",TLBc[2].entries*↩
                sizeof(TLB_ENTRY));
951         //set count variable to 0
952         for(i=0;i<TLBc[2].entries;i++)
953             TLB[2][i].count=0;
954     }
955
956
957 }
958
959
960 /*↩
        ------------------------------------------------------------------------------
        */
961
962 void tlbsim_init(Int tlb_type, Int page_size, Int assoc, Int entries)↩
        {
963
964     //if page_size==-1, then TLB wasn't detected and thus don't do ↩
            anything with it.
965     //VG_(umsg)("tlb type: %d, page_size=%d\n",tlb_type,page_size);
966
967     if(tlb_type==0 && page_size!=-1)
968         sim_iTLB=True;
```

```
969     if(tlb_type==1 && page_size!=-1)
970         sim_dTLB=True;
971     if(tlb_type==2 && page_size!=-1)
972         sim_L2TLB=True;
973 
974     //if(page_size!=-1){
975     TLBc[tlb_type].page_size  = page_size;
976     TLBc[tlb_type].assoc      = assoc;
977     TLBc[tlb_type].entries    = entries;
978     //}
979 }
980 
981 /*↩
        ------------------------------------------------------------------------------------------
        */
982 
983 void print_stats(Int hit, Int miss){
984     VG_(umsg)("Total Accesses:   %d\n",hit+miss);
985     VG_(umsg)("Hits:             %d\n",hit);
986     VG_(umsg)("Misses:           %d\n",miss);
987 
988     static Char buf1[128];
989 
990     //percentify hits
991     VG_(percentify)(hit, hit+miss, 1, 4, buf1);
992     VG_(umsg)("Hit ratio:        %s\n",buf1);
993 
994     //percentify misses
995     VG_(percentify)(miss, hit+miss, 1, 4,buf1);
996     VG_(umsg)("Miss ratio:       %s\n",buf1);
997 
998     VG_(umsg)("\n\n");
999 }
1000 
1001 /*↩
        ------------------------------------------------------------------------------------------
        */
1002 
1003 static void tlb_fini(void)
1004 {
1005 
1006     tlb_chars();
1007     if(clo_sim_tlb){
1008 
1009         if(sim_iTLB){
1010             VG_(umsg)("---iTLB Stats---\n");
1011             print_stats(TLBc[0].hit,TLBc[0].miss);
1012         }
1013 
1014         if(sim_dTLB){
1015             VG_(umsg)("---dTLB Stats---\n");
1016             print_stats(TLBc[1].hit,TLBc[1].miss);
1017         }
1018 
1019         if(sim_L2TLB){
1020             VG_(umsg)("---L2TLB Stats---\n");
1021             print_stats(TLBc[2].hit,TLBc[2].miss);
1022         }
1023 
1024     }
1025 
```

```
1026      if(clo_sim_pages)
1027          print_pages();
1028
1029      //TLB_TYPE=2;
1030      //print_tlb_contents();
1031
1032      //free TLBs
1033      VG_(free)(TLB[0]);
1034      VG_(free)(TLB[1]);
1035      VG_(free)(TLB[2]);
1036      VG_(free)(TLB);
1037
1038  }
1039
1040  /*↩
          ----------------------------------------------------------------------------------------
          */
1041
1042  /*↩
          ------------------------------------------------------------------------↩
          */
1043  /*--- Command line processing                                              ↩
          ---*/
1044  /*↩
          ------------------------------------------------------------------------↩
          */
1045
1046  static Bool tlb_process_cmd_line_option(Char* arg)
1047  {
1048
1049      if      VG_BOOL_CLO(arg, "--tlb-sim"       , clo_sim_tlb  ) {}
1050      else if VG_BOOL_CLO(arg, "--tlb-page-sim"  , clo_sim_pages) {}
1051      else if VG_INT_CLO(arg,  "--tlb-vas-size"  , VAS_SIZE     ) {if↩
            (!(VAS_SIZE>0)) VG_(umsg)("Virtual Address Size has to be ↩
            bigger than 0.\n");tl_assert(VAS_SIZE>0);}
1052      else if VG_INT_CLO(arg,  "--tlb-rep-pol"   , RepPol       ) {if(↩
            RepPol<0 || RepPol>2){ VG_(umsg)("Not valid replacement ↩
            policy value. Setting to LRU.\n");RepPol=1; }}
1053      else
1054          return False;
1055
1056      return True;
1057  }
1058
1059  /*↩
          ----------------------------------------------------------------------------------------
          */
1060
1061  static void tlb_print_usage(void)
1062  {
1063
1064      VG_(printf)(
1065                  // "    TLB simulation Usage\n"
1066                  "    --tlb-sim=yes|no       [yes]     collect TLB ↩
                        stats?\n"
1067                  "    --tlb-page-sim=yes|no  [no]      collect pages ↩
                        used during TLB sim?\n"
1068                  //  "    --enties=<num>          [64]      set TLB's ↩
                        number of entries\n"
1069                  //  "    --assoc=<num>           [Ful Ass] set TLB's ↩
                        associativity, -1 for fully associative, 0 for ↩
```

```
                    direct mapped, N>0 for N-way\n"
1070                //  "    --page-size=<num>      [4096]    set TLB's ↩
                    page size (in bytes)\n"
1071                "    --tlb-vas-size=<num>   [32]      set TLB's ↩
                    virtual address space size (in bits)\n"
1072                "    --tlb-rep-pol=<num>    [1]       set TLB's ↩
                    Replacement Policy 0-> LFU, 1-> LRU, 2->Random\n"
1073                //  "    --type=<num>      [All]          set TLB's ↩
                    type. -1 for all, 0 for L1 iTLB, 1 for L1 dTLB, 2↩
                     for L2TLB\n"
1074                );
1075 }
1076
1077 /*↩
       ------------------------------------------------------------------------↩
       */
1078 /*--- end                                                    cg_tlb.c ---↩
       */
1079 /*↩
       ------------------------------------------------------------------------↩
       */
```